Activiti 5.x User Guide 《Activiti 5.x 用户指南》中文翻译

waylau

Published with GitBook

目錄

activiti-5.x-user-guide

Chinese translation of the Activiti 5.x User Guide . The current version of Activiti is 5.18.0(31 July 2015). This is a GitBook version of the book: http://waylau.gitbooks.io/activiti-5-x-user-guide/. Let's go!

《Activiti 5.x 用户指南》 中文翻译。至今为止，Activiti 的最新版本为 5.18.0(2015-7-31)。利用业余时间对此进行翻译，并在原文的基础上，插入配图，图文并茂方便用户理解。如有勘误欢迎指正，点此提问。如有兴趣，也可以参与到本翻译工作中来 :) 另外有 GitBook 的版本方便阅读http://waylau.gitbooks.io/activiti-5-x-user-guide/

从目录开始阅读吧

Contact:

- Blog:www.waylau.com
- Gmail: waylau521@gmail.com
- Weibo: waylau521
- Twitter: waylau521
- Github : waylau

Chapter 1. Introduction 介绍

License 协议

Activiti 基于 Apache V2 协议.。Activiti Modeler 使用了不同的协议 LGPL 2.1

Download 下载

http://activiti.org/download.html

Sources 源码

发布包里包含大部分的已经打好 jar 包的源码。如果想找到并构建完整的源码库，请参考 wiki 构建发布包

Required software 需要的软件

JDK 6+

JDK 需要 6 或者以上的版本 。在 Oracle Java SE 下载页面 进行下载，安装完成后 执行 `java -version` 命令行来检验是否成功。

Eclipse Indigo and Juno

下载 eclipse 进行安装，版本选 Indigo 或者 Juno。详见 Chapter 12. Eclipse Designer-安装一节

Reporting problems 报告问题

任何一个自觉的开发者都应该看看 如何聪明的提出问题。 看完之后，你可以在用户论坛上进行提问和评论， 或者在JIRA问题跟踪系统 中创建问题。

注意

虽然 *Activiti* 已经托管在 *GitHub* 上了，但是问题不应该提交到*GitHub* 的问题跟踪系统上。如果你想报告一个问题， 不要创建一个*GitHub* 的问题，而是应该使用 *JIRA问题跟踪系统*

译者注：如果对本翻译有疑问，可以在*https://github.com/waylau/activiti-5.x-user-guide/issues* 提问。

Experimental features 试验性功能

标记为 [EXPERIMENTAL] 的章节表示功能尚未稳定。

所有包名中包含 .impl. 的类都是内部实现类，都是不保证稳定的。 不过，如果用户指南把哪些类列为配置项，那么它们可以认为是稳定的。

Internal implementation classes 内部实现类

在 jar 包中，所有包名中包含.impl.（比如：org.activiti.engine.impl.pvm.delegate）的类都是实 现类， 它们应该被视为流程引擎内部的类。对于这些类和接口都不能够保证其稳定性。

One minute version 一分钟版

从 Activiti 网站下载 Activiti Explorer 的 WAR 文件后，可以按照下列步骤以默认配置运行示例。 你需要安装 Java 运行时 和 Apache Tomcat（其实，任何提供了 servlet 功能的 web 容器都可以正常运行。但是我们主要是使用 Tomcat 进行的测试）。

- 把下载的 activiti-explorer.war 复制到 Tomcat 的 webapps 目录下。
- 执行 Tomcat 的 bin 目录下的 startup.bat 或 startup.sh 启动服务器。
- Tomcat 启动后，打开浏览器访问 http://localhost:8080/activiti-explorer（译者注：8080 是你的 Tomcat 安装默认端口，当然你可以给 Tomcat 指定其他端口号）。 使用 kermit/kermit 账号登录。

这样就好了！Activiti Explorer 默认使用 H2 内存数据库，如果你想使用其他数据库 请参考这里长版本-Activiti setup。

Activiti setup 安装

要安装 Activiti， 需要安装 Java 运行时 和 Apache Tomcat同时确保系统变量JAVA_HOME 设置正确。具体方法看你是什么操作系统。

只需要将 WAR 拷贝进 Tomcat 的 webapps 就能运行 Activiti Explorer 和 REST 应用。默认， 应用是运行在内存数据库的，已经包含了示例流程，用户和群组信息。

下面是示例中可以使用的用户：

Table 2.1. The demo users

用户 Id	密码	角色
kermit	kermit	admin
gonzo	gonzo	manager
fozzie	fozzie	user

现在可以用上面的账号访问应用

Table 2.2. The webapp tools

应用名称	URL	描述	
Activiti Explorer	http://localhost:8080/activiti-explorer	用户控制台。使用此工具来启动新的流程，分配任务， 查看和认领任务等， 这个工具还允许对 Activiti 引擎进行管理。	

注意 Activiti Explorer 演示实例只是一种简单快速展示 Activiti 的功能的方式。 但是并不是说只能使用这种方式使用 Activiti。 Activiti 只是一个 jar， 可以内嵌到任何 Java 环境 中：swing 或者 Tomcat, JBoss, WebSphere 等等。 也可以把Activiti 作为一个典型的单独运行的 BPM 服务器运行。 只要 java 可以做的，Activiti也可以。

Activiti database setup 数据库安装

就像在一分钟版本示例里说过的，Activiti Explorer 默认使用 H2 内存数据库。 要让 Activiti 使用独立运行的 H2 数据库或者其他数据库，可以修改 Activiti Explorer web 应用 WEB-INF/ classes 目录下的 db.properties。

另外，注意 Activiti Explorer 自动生成了 demo 用的默认用户和群组，流程定义，数据模型。要想禁用这个功能，要修改 WEB-INF/classes 目录下的 属性文件。 禁用 demo 安装，可以设置所有属性为 false 。从代码中也可以看出，我们可以单独启用或禁用每一项功能。

```
# demo data properties
create.demo.users=true
create.demo.definitions=true
create.demo.models=true
create.demo.reports=true
```

Include the Activiti jar and its dependencies 包含 jar 和依赖

想要包含 jar 和依赖，建议使用 Maven (或者 Ivy) 来简化依赖管理。http://www.activiti.org/community.html#maven.repository 中包含了需要的 jar

如果不想用 Maven，你也可以自己把这些 jar 引入到你的项目中。Activiti 下载 zip 包包含了一个 libs 目录，包含了所有 Activiti的 jar 包（和源代码 jar 包）。依赖没有用这种方式发 布。 Activiti 引擎必须的依赖如下所示（通过 mvn dependency:tree生成）：

```
org.activiti:activiti-engine:jar:5.17.0
+- org.activiti:activiti-bpmn-converter:jar:5.17.0:compile
|  \- org.activiti:activiti-bpmn-model:jar:5.17.0:compile
|     +- com.fasterxml.jackson.core:jackson-core:jar:2.2.3:compile
|     \- com.fasterxml.jackson.core:jackson-databind:jar:2.2.3:compile
|        \- com.fasterxml.jackson.core:jackson-annotations:jar:2.2.3:compile
+- org.activiti:activiti-process-validation:jar:5.17.0:compile
+- org.activiti:activiti-image-generator:jar:5.17.0:compile
+- org.apache.commons:commons-email:jar:1.2:compile
|  +- javax.mail:mail:jar:1.4.1:compile
|  \- javax.activation:activation:jar:1.1:compile
+- org.apache.commons:commons-lang3:jar:3.3.2:compile
+- org.mybatis:mybatis:jar:3.2.5:compile
+- org.springframework:spring-beans:jar:4.0.6.RELEASE:compile
|  \- org.springframework:spring-core:jar:4.0.6.RELEASE:compile
+- joda-time:joda-time:jar:2.6:compile
+- org.slf4j:slf4j-api:jar:1.7.6:compile
+- org.slf4j:jcl-over-slf4j:jar:1.7.6:compile
```

注意：只有使用了 mail service task 才必须引入 mail 依赖jar。

在 Activiti 源代码执行 mvn dependency:copy-dependencie 依赖将会轻松下载

Next steps 下步

使用 Activiti Explorer web 应用 是一个熟悉 Activiti 概念和功能的好办法。但是，Activiti 的主要目标是为你自己的应用添加强大的BPM 和工作流功能。 下面的章节会帮助你熟悉如何在你的环境中使用Activiti 进行编程：

- Chapter 3. Configuration 配置 会教你如何设置 Activiti， 如何获得 ProcessEngine 类的实例，它是所有 Activiti 引擎功能的中心入口。
- Chapter 4. The Activiti API 会带领你了解建立 Activiti API 的服务。 这些服务用简便的方法提供了 Activiti 引擎的强大功能，它们可以使用在任何Java 环境下。
- 对深入了解BPMN 2.0，Activiti 引擎中流程的编写结构感兴趣吗？ 请继续浏览Chapter 7. BPMN 2.0 Introduction 介绍 BPMN 2.0

Creating a ProcessEngine 创建 ProcessEngine

Activiti 流程引擎的配置文件是名为 activiti.cfg.xml 文件。 注意这与使用 Spring 方式创建流程引擎是不一样的。

获得 ProcessEngine 最简单的办法是 使用org.activiti.engine.ProcessEngines 类：

```
ProcessEngine processEngine = ProcessEngines.getDefaultProcessEngine()
```

它会在 classpath 下搜索 activiti.cfg.xml， 并基于这个文件中的配置构建引擎。 下面代码展示了实例配置。 后面的章节会给出配置参数的详细介绍。

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans   http://www.springframework.org/schema/beans/sp

  <bean id="processEngineConfiguration" class="org.activiti.engine.impl.cfg.StandaloneProcessEngineConfiguration">

    <property name="jdbcUrl" value="jdbc:h2:mem:activiti;DB_CLOSE_DELAY=1000" />
    <property name="jdbcDriver" value="org.h2.Driver" />
    <property name="jdbcUsername" value="sa" />
    <property name="jdbcPassword" value="" />

    <property name="databaseSchemaUpdate" value="true" />

    <property name="jobExecutorActivate" value="false" />
    <property name="asyncExecutorEnabled" value="true" />
    <property name="asyncExecutorActivate" value="false" />

    <property name="mailServerHost" value="mail.my-corp.com" />
    <property name="mailServerPort" value="5025" />
  </bean>

</beans>
```

注意配置 XML 文件其实是一个 Spring 的配置文件。 但不是说 **Activiti** 只能用在 **Spring** 环境 中！ 我们只是利用了 Spring 的解析和依赖注入功能 来构建引擎。

配置文件中使用的 ProcessEngineConfiguration 可以通过编程方式创建。 可以使用不同的 bean id（比如，例子第三行）。

```
ProcessEngineConfiguration.createProcessEngineConfigurationFromResourceDefault();
ProcessEngineConfiguration.createProcessEngineConfigurationFromResource(String resource);
ProcessEngineConfiguration.createProcessEngineConfigurationFromResource(String resource, String beanName);
ProcessEngineConfiguration.createProcessEngineConfigurationFromInputStream(InputStream inputStream);
ProcessEngineConfiguration.createProcessEngineConfigurationFromInputStream(InputStream inputStream, String beanName);
```

也可以不使用配置文件，基于默认创建配置 （参考各种支持类）

```
ProcessEngineConfiguration.createStandaloneProcessEngineConfiguration();
ProcessEngineConfiguration.createStandaloneInMemProcessEngineConfiguration();
```

所有这些 ProcessEngineConfiguration.createXXX() 方法都返回 ProcessEngineConfiguration，后续可以调整成所需的对象。 在调用 buildProcessEngine() 后， 就会创建一个 ProcessEngine：

```
ProcessEngine processEngine = ProcessEngineConfiguration.createStandaloneInMemProcessEngineConfiguration()
  .setDatabaseSchemaUpdate(ProcessEngineConfiguration.DB_SCHEMA_UPDATE_FALSE)
  .setJdbcUrl("jdbc:h2:mem:my-own-db;DB_CLOSE_DELAY=1000")
  .setAsyncExecutorEnabled(true)
  .setAsyncExecutorActivate(false)
  .buildProcessEngine();
```

ProcessEngineConfiguration bean

activiti.cfg.xml 必须包含一个 bean, id为'processEngineConfiguration'。

```
<bean id="processEngineConfiguration" class="org.activiti.engine.impl.cfg.StandaloneProcessEngineConfiguration">
```

这个 bean 会用来构建 ProcessEngine。 有多个类可以用来定义processEngineConfiguration。 这些类对应不同的环境，并设置了对应的默认值。 最好选择（最）适用于你的环境的类， 这样可以少配置几个引擎的参数。 下面是目前可以使用的类（以后会包含更多）：

- org.activiti.engine.impl.cfg.StandaloneProcessEngineConfiguration: 单独运行的 流程引擎。Activiti 会自己处理事务。 默认，数据库只在引擎启动时检测 （如果没有 Activiti 的表或者表结构不正确就会抛出异常）。
- org.activiti.engine.impl.cfg.StandaloneInMemProcessEngineConfiguration: 单元测试时的辅助类。Activiti 会自己控制事务。 默认使用 H2 内存数据库。数据库表会在引擎启动时创建，关闭时删除。 使用它时，不需要其他配置（除非使用 job 执行器或邮件功 能）。
- org.activiti.spring.SpringProcessEngineConfiguration: 在Spring 环境下使用流程引擎。 参考 Chapter 5. Spring integration 集成 Spring。
- org.activiti.engine.impl.cfg.JtaProcessEngineConfiguration: 单独运行流程引擎，并使用 JTA 事务。

Database configuration 数据库配置

两种方式配置数据库给 Activiti 引擎使用。首先是定义数据库 JDBC 属性：

- jdbcUrl: 数据库的 JDBC URL
- jdbcDriver: 数据库类型的驱动实现
- jdbcUsername: 数据库连接用户名
- jdbcPassword: 数据库连接密码

基于 JDBC 参数配置的数据库连接 会使用默认的MyBatis连接池。 下面的参数可以用来配置连接池（来自MyBatis参数）：

- jdbcMaxActiveConnections: 连接池中处于被使用状态的连接的最大值。默认为10
- jdbcMaxIdleConnections: 连接池中处于空闲状态的连接的最大值
- jdbcMaxCheckoutTime: 连接被取出使用的最长时间，超过时间会被强制回收。 默认为20000（20秒）
- jdbcMaxWaitTime: 这是一个底层配置，让连接池可以在长时间无法获得连接时， 打印一条日志，并重新尝试获取一个连接。（避免因为错误配置导致沉默的操作失败）。 默认为20000（20秒）。

数据配置示例:

```
<property name="jdbcUrl" value="jdbc:h2:mem:activiti;DB_CLOSE_DELAY=1000" />
<property name="jdbcDriver" value="org.h2.Driver" />
<property name="jdbcUsername" value="sa" />
<property name="jdbcPassword" value="" />
```

也可以使用 javax.sql.DataSource 实现 （比如，Apache Commons 的 DBCP）：

```
<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource" >
 <property name="driverClassName" value="com.mysql.jdbc.Driver" />
 <property name="url" value="jdbc:mysql://localhost:3306/activiti" />
 <property name="username" value="activiti" />  <property name="password" value="activiti" />  <property name="defaultA
</bean>
<bean id="processEngineConfiguration" class="org.activiti.engine.impl.cfg.StandaloneProcessEngineConfiguration">
 <property name="dataSource" ref="dataSource" />    ...
```

注意，Activiti 的发布包中没有这些类。 你要自己把对应的类（比如，从 DBCP 里）放到你的 classpath 下。 无论你使用 JDBC 还是 DataSource 的方式，都可以设置下面的配置：

- databaseType: 一般不用设置，因为可以自动通过数据库连接的元数据获取。 只有自动检测失败时才需要设置。 可能的值有：{h2, mysql, oracle, postgres, mssql, db2}。 如果没使用默认的**H2**数据库就必须设置这项。 这个配置会决定使用哪些创建/删除脚本和查询语句。 参考支持数据库章节 了解支持哪些类型。
- databaseSchemaUpdate: 设置流程引擎启动和关闭时如何处理数据库表。
    - false（默认）：检查数据库表的版本和依赖库的版本， 如果版本不匹配就抛出异常。
    - true: 构建流程引擎时，执行检查，如果需要就执行更新。 如果表不存在，就创建
    - create-drop: 构建流程引擎时创建数据库表， 关闭流程引擎时删除这些表。

JNDI Datasource Configuration 数据源配置

默认，Activiti 的数据库配置会放在 web 应用的 WEB-INF/classes 目录下的 db.properties 文件中。 这样做比较繁琐， 因为要用户在每次发布时，都修改 Activiti 源码中的 db.properties 并重新编译 war 文件， 或者解压缩 war 文件，修改其中的 db.properties。

使用 JNDI（Java Naming and Directory Interface - Java 命名和目录接口）来获取数据库连接， 连接是由 Servlet 容器管理的，可以 在 war 部署外边管理配置。 与 db.properties 相比， 它也允许对连接进行更多的配置。

Usage 使用

要想把 Activiti Explorer 和 Activiti Rest 应用从db.properties 转换为使用 JNDI 数据库配 置，需要打开原始的 Spring 配置文件 （activiti-webapp-explorer2/src/main/webapp/WEBINF/activiti-standalone-context.xml 和activiti-webapp-rest2/src/main/resources/ activiti-context.xml）， 删除"dbProperties"和"dataSource"两个bean，然后添加如下 bean：

```
<bean id="dataSource" class="org.springframework.jndi.JndiObjectFactoryBean">
<property name="jndiName" value="java:comp/env/jdbc/activitiDB"/>
</bean>
```

接下来，我们需要添加包含了默认的 H2 配置的 context.xml 文件。 如果已经有了 JNDI 配置， 会覆盖这些配置。 对 Activiti Explorer 来说，对应的配置文件 activiti-webapp-explorer2/src/main/webapp/META-INF/context.xml 如下所示：

```
<Context antiJARLocking="true" path="/activiti-explorer2">
    <Resource auth="Container"
              name="jdbc/activitiDB"
              type="javax.sql.DataSource"
              scope="Shareable"
              description="JDBC DataSource"
              url="jdbc:h2:mem:activiti;DB_CLOSE_DELAY=1000"
              driverClassName="org.h2.Driver"
              username="sa"
              password=""
              defaultAutoCommit="false"
              initialSize="5"
              maxWait="5000"
              maxActive="120"
              maxIdle="5"/>
</Context>
```

对于 Activiti REST web 应用，添加的 activiti-webapp-rest2/src/main/webapp/META-INF/context.xml 如下所示：

```
<?xml version="1.0" encoding="UTF-8"?>
<Context antiJARLocking="true" path="/activiti-rest2">
    <Resource auth="Container"
              name="jdbc/activitiDB"
              type="javax.sql.DataSource"
              scope="Shareable"
              description="JDBC DataSource"
              url="jdbc:h2:mem:activiti;DB_CLOSE_DELAY=-1"
              driverClassName="org.h2.Driver"
              username="sa"
              password=""
              defaultAutoCommit="false"
              initialSize="5"
              maxWait="5000"
              maxActive="120"
              maxIdle="5"/>
</Context>
```

可选的一步，现在可以删除 Activiti Explorer 和 Activiti REST 两个应用中 不再使用的 db.properties 文件了。

Configuration 配置

JNDI 数据库配置会因为你使用的 Servlet 容器 不同而不同。 下面的配置可以在 Tomcat 中使用，但是对其他容器，请引用你使用的容器的文档。

如果使用 Tomcat，JNDI资源配置在 $CATALINA_BASE/conf/[enginename]/[hostname]/[warname].xml （对于 Activiti Explorer 来说，通常是在 $CATALINA_BASE/conf/ Catalina/localhost/activiti-explorer.war）。 当应用第一次发布时，会把这个文件从 war 中复制出来。 所以如果这个文件已经存在了，你需要替换它。要想修改 JNDI 资源让应用连接 mysql 而不是 H2，可以像下面这样修改：

```
<?xml version="1.0" encoding="UTF-8"?>
    <Context antiJARLocking="true" path="/activiti-explorer2">
        <Resource auth="Container"
            name="jdbc/activitiDB"
            type="javax.sql.DataSource"
            description="JDBC DataSource"
            url="jdbc:mysql://localhost:3306/activiti"
            driverClassName="com.mysql.jdbc.Driver"
            username="sa"
            password=""
            defaultAutoCommit="false"
            initialSize="5"
            maxWait="5000"
            maxActive="120"
            maxIdle="5"/>
        </Context>
```

Supported databases 支持的数据库

下面是可以引用的类型

Table 3.1. Supported databases

Activiti 数据库类型	JDBC URL示例	备注
h2	jdbc:h2:tcp://localhost/activiti	默认配置数据库
mysql	jdbc:mysql://localhost:3306/activiti?autoReconnect=true	使用 mysql-connector-java 驱动进行测试
oracle	jdbc:oracle:thin:@localhost:1521:xe	
postgres	jdbc:postgresql://localhost:5432/activiti	
db2	jdbc:db2://localhost:50000/activiti	
mssql	jdbc:sqlserver://localhost:1433/activiti	

Creating the database tables 创建数据库表

下面是创建数据库表最简单的办法：

- 把 activiti-engine 的 jar 放到 classpath 下
- 添加对应的数据库驱动
- 把 Activiti 配置文件 (activiti.cfg.xml) 放到 classpath下， 指向你的数据库（参考数据库配置章节）
- 执行 DbSchemaCreate 类的 main 方法

不过，一般情况只有数据库管理员才能执行 DDL 语句。 在生产环境，这也是最明智的选择。SQL DDL语句可以从 Activiti 下载页或 Activiti发布目录里找到，在 database 子目录下。 脚本 也包含在引擎的 jar 中(activiti-engine-x.jar)， 在 org/activiti/db/create包下（drop 目录里是删除语句）。 SQL 文件的命名方式如下

```
activiti.{db}.{create|drop}.{type}.sql
```

其中 db 是 支持的数据库， type 是

- engine: 引擎执行的表。必须。
- identity: 包含用户，群组，用户与组之间的关系的表。 这些表是可选的，只有使用引擎自带的默认身份管理时才需要。
- history: 包含历史和审计信息的表。可选的：历史级别设为 none 时不会使用。 注意这也会引用一些需要把数据保存到历史表中的功能（比如任务的评论）。

MySQL用户需要注意： 版本低于 5.6.4 的 MySQL 不支持毫秒精度的 timestamp 或 date 类型。 更严重的是，有些版本会在尝试创建这样一列时抛出异常，而有些版本则不会。 在执行自动创建/更新时，引擎会在执行过程中修改 DDL。 当使用 DDL时，可以选择通用版本和名为 mysql55的文件。 （它适合所有版本低于5.6.4的情况）。 后一个文件会将列的类型设置为没有毫秒的情况。

总结一下，对于MySQL版本会执行如下操作

- <5.6: 不支持毫秒精度。可以使用 DDL 文件（包含 mysql55 的文件）。可以实现自动创建/更新。
- 5.6.0 - 5.6.3: 不支持毫秒精度。无法自动创建/更新。建议更新到新的数据库版本。如果真的需要的话，也可以使用 mysql 5.5。
- 5.6.4+:支持毫秒精度。可以使用DDL文件（默认包含 mysql 的文件）。可以实现自动创建、更新。

注意对于已经更新了 MySQL 数据库，而且 Activiti 表已经创建/更新的情况， 必须手工修改列的类型。

Database table names explained 理解数据库表名字

Activiti 的表都以 ACT_ 开头。 第二部分是表示表的用途的两个字母标识。 用途也和服务的 API 对应。

- ACT*RE**: 'RE'表示 repository。 这个前缀的表包含了流程定义和流程静态资源 （图片，规则， 等等）。
- ACT*RU**: 'RU'表示 runtime。 这些运行时的表，包含流程实例，任务， 变量，异步任务，等运行中的数据。 Activiti 只在流程实例执行过程中保存这些数据， 在流程结束时就会删除这些记录。 这样运行时表可以一直很小速度很快。
- ACT*ID**: 'ID'表示 identity。 这些表包含身份信息，比如用户， 组等等。
- ACT*HI**: 'HI'表示 history。 这些表包含历史数据，比如历史流程实例， 变量，任务等等。
- ACT*GE**: general 数据， 用于不同场景下。

Database upgrade 数据库升级

在执行更新之前要先备份数据库（使用数据库的备份功能）

默认，每次构建流程引擎时都会进行版本检测。 这一切都在应用启动或Activiti webapp 启动时发生。 如果 Activiti 发现数据库表的版本与依赖库的版本不同， 就会抛出异常。

要升级，你要把下面的配置 放到 activiti.cfg.xml 配置文件里：

```
<beans ... >

  <bean id="processEngineConfiguration" class="org.activiti.engine.impl.cfg.StandaloneProcessEngineConfiguration">
    <!-- ... -->
    <property name="databaseSchemaUpdate" value="true" />
    <!-- ... -->
  </bean>

</beans>
```

然后，把对应的数据库驱动放到 **classpath** 里。 升级应用的 Activiti依赖。启动一个新版本的 Activiti 指向包含旧版本的数据库。将 databaseSchemaUpdate 设置为 true， Activiti 会自动将 数据库表升级到新版本,

当发现依赖和数据库表版本不通过时。也可以执行更新升级 DDL 语句。 也可以执行数据库脚本，可以在 Activiti 下载页找到。

Job Executor and Async Executor (since version 5.17.0)

从版本 5.17.0 开始，除了 Job Executor（作业执行器）之外， Activiti 还提供了一个 Async executor （异步执行器）。Async executor 在Activiti 引擎中 是一个更好的性能和对数据库更友好的执行异步作业的方式。因此建议切换到 Async executor，在默认情况下仍然使用旧的 job executor 。更多的信息可以在用户指南的高级部分找到。

Job executor activation 启用 Job executor

JobExecutor 是管理一系列线程的组件，可以触发定时器（也包含后续的异步消息）。 在单元测试场景下，很难使用多线程。因此 API 允许查询(ManagementService.createJobQuery) 和执行 job (ManagementService.executeJob)，所以 job 可以在单元测试中控制。 要避免与 job 执行器冲突，可以关闭它。

默认，JobExecutor 在流程引擎启动时就会激活。 如果不想在流程引擎启动后自动激活 JobExecutor，可以设置

```
<property name="jobExecutorActivate" value="false" />
```

当流程引擎引导时,不想激活 JobExecutor。

Async executor activation 启用 Async executor

AsyncExecutor 是管理线程池的组件，可以触发定时器和异步任务。

默认，AsyncExecutor 是不启用的，由于遗留原因使用的是 JobExecutor。不过建议使用新的 AsyncExecutor 来代替。可以通过定义两个属性

```
<property name="asyncExecutorEnabled" value="true" />
<property name="asyncExecutorActivate" value="true" />
```

asyncExecutorEnabled 属性启用 Async executor 代替旧的 Job executor。第二个属性 asyncExecutorActivate 指示 Activiti 引擎在启动时启动 Async executor 线程池。

Mail server configuration 配置邮件服务器

可以选择配置邮件服务器。Activiti 支持在业务流程中发送邮件。 想真正的发送一个 e-mail，必须配置一个真实的 SMTP 邮件服务器。 参考 e-mail 任务。

History configuration 配置历史

可以选择定制历史存储的配置。你可以通过配置影响引擎的历史功能。 参考 History configuration 配置这一节。

```
<property name="history" value="audit" />
```

Exposing configuration beans in expressions and scripts 在表达式和脚本中暴露配置

默认，activiti.cfg.xml 和你自己的 Spring 配置文件中所有 bean 都可以在表达式和脚本中使用。 如果你想限制配置文件中的 bean 的可见性， 可以配置流程引擎配置的 bean 配置。 ProcessEngineConfiguration 的 beans 是一个 map。当你指定了这个参数， 只有包含这个 map 中的 bean 可以在表达式和脚本中使用。 通过在map 中指定的名称来决定暴露的 bean。

Deployment cache configuration 配置部署缓存

所有流程定义都被缓存了（解析之后）避免每次使用前都要访问数据库， 因为流程定义数据是不会改变的。 默认，不会限制这个缓存。如果想限制流程定义缓存，可以添加如下配置

```
<property name="processDefinitionCacheLimit" value="10" />
```

这个配置会把默认的 hashmap 缓存替换成 LRU 缓存，来提供限制。 当然，这个配置的最佳值跟流程定义的总数有关， 实际使用中会具体使用多少流程定义也有关。 也你可以注入自己的缓存实现。这个 bean 必须实现 org.activiti.engine.impl.persistence.deploy.DeploymentCache 接口：

```
<property name="processDefinitionCache">
  <bean class="org.activiti.MyCache" />
</property>
```

有一个类似的配置叫 knowledgeBaseCacheLimit 和knowledgeBaseCache， 它们是配置规则缓存的。只 有流程中使用规则任务时才会用到。

Logging 日志

从 Activiti 5.12 开始，SLF4J 被用作日志框架，替换了之前使用java.util.logging。 所有日志（activiti, spring, mybatis等等）都转发给 SLF4J 允许使用你选择的日志实现。

默认 **activiti-engine** 依赖中没有提供 **SLF4J** 绑定的**jar**， 需要根据你的实际需要使用日志框架。如果没有添加任何实现jar，SLF4J 会使用NOP-logger，不使用任何日志，不会发出警告，而且什么日志都不会记录。 可以通过 http://www.slf4j.org/codes.html#StaticLoggerBinder 了解这些实现。 使用 Maven，比如使用一个依赖（这里使用 log4j），注意你还需要添加一个 version： 配置

org.slf4j slf4j-log4j12
activiti-explorer 和 activiti-rest 应用都使用了 Log4j 绑定。执行所有 activiti-* 模块的单元测试页使用了 Log4j。

特别提醒如果容器 **classpath** 中存在 **commons-logging**： 为了把spring 日志转发给 SLF4J，需要使用桥接（参考 http://www.slf4j.org/legacy.html#jclOverSLF4J）。 如果你的容器提 供了commons-logging实现，请参考下面网页： http://www.slf4j.org/codes.html#release 来确保稳定性。 使用 Maven 的实例（忽略版本）：

```
<dependency>
  <groupId>org.slf4j</groupId>
  <artifactId>slf4j-log4j12</artifactId>
</dependency>
```

activiti-explorer 和 activiti-rest 应用都使用了 Log4j 绑定。执行所有 activiti-* 模块的单元测试页使用了 Log4j。

特别提醒如果容器 **classpath** 中存在 **commons-logging**： 为了把spring 日志转发给 SLF4J，需要使用桥接（参考 http://www.slf4j.org/legacy.html#jclOverSLF4J）。 如果你的容器提供了 commons-logging 实现，请参考下面网页：http://www.slf4j.org/codes.html#release 来确保稳定性。

使用 Maven 的实例（忽略版本）：

```
<dependency>
  <groupId>org.slf4j</groupId>
  <artifactId>jcl-over-slf4j</artifactId>
</dependency>
```

Mapped Diagnostic Contexts 映射诊断上下文

在5.13中，activiti 支持 slf4j 的 MDC 功能。 如下的基础信息会传递到日志中记录：

- 流程定义 Id 标记为 mdcProcessDefinitionID
- 流程实例 Id 标记 为mdcProcessInstanceID
- 分支 Id 标记为 mdcexecutionId

默认不会记录这些信息。可以配置日志使用期望的格式来显示它们，扩展通常的日志信息。比如，下面的 log4j 配置定义会让日志显示上面提及的信息：

```
log4j.appender.consoleAppender.layout.ConversionPattern =ProcessDefinitionId=%X{mdcProcessDefinitionID}
executionId=%X{mdcExecutionId} mdcProcessInstanceID=%X{mdcProcessInstanceID} mdcBusinessKey=%X{mdcBusinessKey} %m%n"
```

当系统进行高风险任务，日志必须严格检查时，这个功能就非常有用，比如要使用日志分析的情况。

Event handlers 事件处理

Activiti 5.15 中实现了一种事件机制。它允许在引擎触发事件时获得提醒。 参考所有支持的事件类型了解有效的事件。

可以为对应的事件类型注册监听器，在这个类型的任何时间触发时都会收到提醒。 你可以添加引擎范围的事件监听器通过配置， 添加引擎范围的事件监听器在运行阶段使用API， 或添加 event-listener 到特定流程定义的 BPMN XML 中。

所有分发的事件，都是org.activiti.engine.delegate.event.ActivitiEvent 的子类。事件包含（如果有效）type，executionId，processInstanceId 和processDefinitionId。 对应的事件会包含事件发生时对应上下文的额外信息， 这些额外的载荷可以在所有支持的事件类型中找到。

Event listener implementation 事件监听实现

实现事件监听器的唯一要求是实现org.activiti.engine.delegate.event.ActivitiEventListener。 西面是一个实现监听器的例子，它会把所有监听到的事件打印到标准输出中，包括job执行的事件异常：

```
public class MyEventListener implements ActivitiEventListener {

  @Override
  public void onEvent(ActivitiEvent event) {
    switch (event.getType()) {

      case JOB_EXECUTION_SUCCESS:
        System.out.println("A job well done!");
        break;

      case JOB_EXECUTION_FAILURE:
        System.out.println("A job has failed...");
        break;

      default:
        System.out.println("Event received: " + event.getType());
    }
  }

  @Override
  public boolean isFailOnException() {
    // The logic in the onEvent method of this listener is not critical, exceptions
    // can be ignored if logging fails...
    return false;
  }
}
```

isFailOnException() 方法决定了当事件分发时，onEvent(..) 方法抛出异常时的行为。 这里返回的是 false，会忽略异常。 当返回 true 时，异常不会忽略，继续向上传播，迅速导致当前命令失败。 当事件是一个 API 调用的一部分时（或其他事务性操作，比如 job 执行）， 事务就会回滚。当事件监听器中的行为不是业务性时，建议返回 false。 activiti 提供了一些基础的实现， 实现了事件监听器的常用场景。可以用来作为基类或监听器实现的样例：

- org.activiti.engine.delegate.event.BaseEntityEventListener： 这个事件监听器的基 类可以用来监听实体相关的事件，可以针对某一类型实体，也可以是全部实体。 它隐藏了 类型检测，并提供了三个需要重写的方法：onCreate(..), onUpdate(..) 和 onDelete(..)，当实 体创建，更新，或删除时调用。对于其他实体相关的事件，会调用 onEntityEvent(..)。

Configuration and setup 配置和安装

把事件监听器配置到流程引擎配置中时，会在流程引擎启动时激活，并在引擎启动启动中持续工作着。

eventListeners 属性需要org.activiti.engine.delegate.event.ActivitiEventListener 的队列。 通常，我们可以声明一个内部的bean定义，或使用 ref 引用已定义的 bean。 下面的代码，向配置添加了一个事件监听器，任何事件触发时都会提醒它，无

论事件是什么类型：

```
<bean id="processEngineConfiguration" class="org.activiti.engine.impl.cfg.StandaloneProcessEngineConfiguration">
    ...
    <property name="eventListeners">
      <list>
         <bean class="org.activiti.engine.example.MyEventListener" />
      </list>
    </property>
</bean>
```

为了监听特定类型的事件，可以使用 typedEventListeners 属性，它需要一个 map 参数。 map 的 key 是逗号分隔的事件名（或单独的事件名）。 map 的 value 是 org.activiti.engine.delegate.event.ActivitiEventListener 队列。 下面的代码演示了向配置中添加一个事件监听器，可以监听 job 执行成功或失败：

```
<bean id="processEngineConfiguration" class="org.activiti.engine.impl.cfg.StandaloneProcessEngineConfiguration">
    ...
    <property name="typedEventListeners">
      <map>
        <entry key="JOB_EXECUTION_SUCCESS,JOB_EXECUTION_FAILURE" >
          <list>
            <bean class="org.activiti.engine.example.MyJobEventListener" />
          </list>
        </entry>
      </map>
    </property>
</bean>
```

分发事件的顺序是由监听器添加时的顺序决定的。首先，会调用所有普通的事件监听器（eventListeners 属性），按照它们在 list 中的次序。 然后，会调用所有对应类型的监听器（typedEventListeners 属性），如果对应类型的事件被触发了。

Adding listeners at runtime 运行时添加监听

可以通过API（RuntimeService）在运行阶段添加或删除额外的事件监听器

```
/**
 * Adds an event-listener which will be notified of ALL events by the dispatcher.
 * @param listenerToAdd the listener to add
 */
void addEventListener(ActivitiEventListener listenerToAdd);

/**
 * Adds an event-listener which will only be notified when an event occurs, which type is in the given types.
 * @param listenerToAdd the listener to add
 * @param types types of events the listener should be notified for
 */
void addEventListener(ActivitiEventListener listenerToAdd, ActivitiEventType... types);

/**
 * Removes the given listener from this dispatcher. The listener will no longer be notified,
 * regardless of the type(s) it was registered for in the first place.
 * @param listenerToRemove listener to remove
 */
 void removeEventListener(ActivitiEventListener listenerToRemove);
```

注意运行期添加的监听器引擎重启后就消失了

Adding listeners to process definitions 添加监听到流程定义

可以为特定流程定义添加监听器。监听器只会监听与这个流程定义相关的事件，以及这个流程定义上发起的所有流程实例的事件。 监听器实现可以使用，全类名定义，引用实现了监听器接口的表达式，或配置为抛出一个message/signal/error 的

BPMN 事件。

Listeners executing user-defined logic 让监听器执行用户定义的逻辑

下面代码为一个流程定义添加了两个监听器。第一个监听器会接收所有类型的事件，它是通过全类名定义的。 第二个监听器只接收作业成功或失败的事件，它使用了定义在流程引擎配置中的 beans 属性中的一个 bean。

```
<process id="testEventListeners">
  <extensionElements>
    <activiti:eventListener class="org.activiti.engine.test.MyEventListener" />
    <activiti:eventListener delegateExpression="${testEventListener}" events="JOB_EXECUTION_SUCCESS,JOB_EXECUTION_FAILU
  </extensionElements>

  ...

</process>
```

对于实体相关的事件，也可以设置为针对某个流程定义的监听器，实现只监听发生在某个流程定义上的某个类型实体事件。 下面的代码演示了如何实现这种功能。可以用于所有实体事件（第一个例子），也可以只监听特定类型的事件（第二个例子）

```
<process id="testEventListeners">
  <extensionElements>
    <activiti:eventListener class="org.activiti.engine.test.MyEventListener" entityType="task" />
    <activiti:eventListener delegateExpression="${testEventListener}" events="ENTITY_CREATED" entityType="task" />
  </extensionElements>

  ...

</process>
```

entityType 支持的值有：attachment, comment, execution,identity-link, job, process-instance, process-definition, task。

```
<process id="testEventListeners">
  <extensionElements>
    <activiti:eventListener class="org.activiti.engine.test.MyEventListener" entityType="task" />
    <activiti:eventListener delegateExpression="${testEventListener}" events="ENTITY_CREATED" entityType="task" />
  </extensionElements>

  ...

</process>
```

Listeners throwing BPMN events

[试验阶段]

另一种处理事件的方法是抛出一个 BPMN 事件。请注意它只针对与抛出一个activiti 事件类型的 BPMN 事件。 比如，抛出一个 BPMN 事件，在流程实例删除时，会导致一个错误。 下面的代码演示了如何在流程实例中抛出一个 signal，把 signal 抛出到外部流程（全局），在流程实例中抛出一个消息事件，在流程实例中抛出一个错误事件。除了使用 class 或 delegateExpression， 还使用了 throwEvent 属性，通过额外属性，指定了抛出事件的类型。

```
    <process id="testEventListeners">
      <extensionElements>
        <activiti:eventListener throwEvent="signal" signalName="My signal" events="TASK_ASSIGNED" />
      </extensionElements>
    </process>
```

```
<process id="testEventListeners">
  <extensionElements>
    <activiti:eventListener throwEvent="globalSignal" signalName="My signal" events="TASK_ASSIGNED" />
  </extensionElements>
</process>
```

```
<process id="testEventListeners">
  <extensionElements>
    <activiti:eventListener throwEvent="message" messageName="My message" events="TASK_ASSIGNED" />
  </extensionElements>
</process>
```

```
<process id="testEventListeners">
  <extensionElements>
    <activiti:eventListener throwEvent="error" errorCode="123" events="TASK_ASSIGNED" />
  </extensionElements>
</process>
```

如果需要声明额外的逻辑，是否抛出 BPMN 事件，可以扩展 activiti 提供的监听器类。在子类中重写 isValidEvent(ActivitiEvent event)， 可以防止抛出 BPMN 事件。对应的类是 org.activiti.engine.test.api.event.SignalThrowingEventListenerTest, org.activiti.engine.impl.bpmn.helper.MessageThrowingEventListener 和 org.activiti.engine.impl.bpmn.helper.ErrorThrowingEventListener.

Notes on listeners on a process-definition 流程定义中监听器的注意事项

- 事件监听器只能声明在 process 元素中，作为 extensionElements 的子元素。 监听器不能定义在流程的单个 activity 下。
- delegateExpression 中的表达式无法访问 execution 上下文，这与其他表达式不同（比如 gateway ）。 它只能引用定义在流程引擎配置的beans 属性中声明的 bean，或者使用 spring（未使用 beans 属性）中所有实现了监听器接口的 spring-bean。
- 在使用监听器的 class 属性时，只会创建一个实例。记住监听器实现不会依赖成员变量，确认是多线程安全的。
- 当一个非法的事件类型用在 events 属性或 throwEvent 中时，流程定义发布时就会抛出异常。（会导致部署失败）。如果 class 或delegateExecution 由问题（类不存在，不存在的 bean 引用，或代理类没有实现监听器接口），会在流程启动时抛出异常（或在第一个有效的流程定义事件被监听器接收时）。所以要保证引用的类正确的放在 classpath 下，表达式也要引用一个有效的实例。

Dispatching events through API 通过 API 分发事件

我们提供了通过 API 使用事件机制的方法，允许大家触发定义在引擎中的任何自定义事件。 建议（不强制）只触发类型为 CUSTOM 的 ActivitiEvents。可以通过RuntimeService 触发事件：

```
/**
 * Dispatches the given event to any listeners that are registered.
 * @param event event to dispatch.
 *
 * @throws ActivitiException if an exception occurs when dispatching the event or when the {@link ActivitiEventDispatch
 * is disabled.
 * @throws ActivitiIllegalArgumentException when the given event is not suitable for dispatching.
 */
 void dispatchEvent(ActivitiEvent event);
```

Supported event types 支持的事件类型

下面是引擎中可能出现的所有事件类型。每个类型都对应 org.activiti.engine.delegate.event.ActivitiEventType 中的一个枚举值

Table 1. Supported events

Event 名称	描述	Event 类
ENGINE_CREATED	监听器监听的流程引擎已经创建 完毕，并准备好接受API调用。	`org.activiti...ActivitiEvent`
ENGINE_CLOSED	监听器监听的流程引擎已经关 闭，不再接受API调用。	`org.activiti...ActivitiEvent`
ENTITY_CREATED	创建了一个新实体。实体包含在 事件中。	`org.activiti...ActivitiEntityEvent`
ENTITY_INITIALIZED	创建了一个新实体，初始化也完成了。如果这个实体的创建会包含子实体的创建，这个事件会在子实体都创建/初始化完成后被触发，这是与 ENTITY_CREATED的区别。 `ENTITY_CREATE` event.	`org.activiti...ActivitiEntityEvent`
ENTITY_UPDATED	更新了已存在的实体。实体包含 在事件中。	`org.activiti...ActivitiEntityEvent`
ENTITY_DELETED	删除了已存在的实体。实体包含 在事件中。	`org.activiti...ActivitiEntityEvent`
ENTITY_SUSPENDED	暂停了已存在的实体。实体包含 在事件中。会被 ProcessDefinitions,ProcessInstances 和 Tasks抛出。	`org.activiti...ActivitiEntityEvent`
ENTITY_ACTIVATED	激活了已存在的实体，实体包含 在事件中。会被 ProcessDefinitions, ProcessInstances 和 Tasks抛出。	`org.activiti...ActivitiEntityEvent`
JOB_EXECUTION_SUCCESS	作业执行成功。job包含在事件中。	`org.activiti...ActivitiEntityEvent`
JOB_EXECUTION_FAILURE	作业执行失败。作业和异常信息包含在事件中。	`org.activiti...ActivitiEntityEvent` and `org.activiti...ActivitiExceptionEve`
JOB_RETRIES_DECREMENTED	因为作业执行失败，导致重试次 数减少。作业包含在事件中。	`org.activiti...ActivitiEntityEvent`
TIMER_FIRED	触发了定时器。job包含在事件 中。	`org.activiti...ActivitiEntityEvent`
JOB_CANCELED	取消了一个作业。事件包含取消 的作业。作业可以通过API调用取消， 任务完成后对应的边界定时器也会取消，在新流程定义发布时也会取消。	`org.activiti...ActivitiEntityEvent`
ACTIVITY_STARTED	一个节点开始执行	`org.activiti...ActivitiActivityEver`
ACTIVITY_COMPLETED	一个节点成功结束	`org.activiti...ActivitiActivityEver`
ACTIVITY_CANCELLED	要取消一个节点。取消是因为三个原因（MessageEventSubscriptionEntity, SignalEventSubscriptionEntity, TimerEntity）	`org.activiti… ActivitiActivityCancelledEvent`
ACTIVITY_SIGNALED	一个节点收到了一个信号	`org.activiti...ActivitiSignalEvent`
	一个节点收到了一个消息。在节点收到消息之前触发。收到后，会触发	

ACTIVITY_MESSAGE_RECEIVED	ACTIVITY_SIGNAL 或 ACTIVITY_STARTED，这会根据节点的类型（边界事件，事件子流程开始事件）	
ACTIVITY_ERROR_RECEIVED	一个节点收到了一个错误事件。在节点实际处理错误之前触发。事件的activityId对应着处理错误的节点。 这个事件后续会是ACTIVITY_SIGNALLED或ACTIVITY_COMPLETE，如果错误发送成功的话。	
UNCAUGHT_BPMN_ERROR	抛出了未捕获的BPMN错误。流程 没有提供针对这个错误的处理器。 事件的activityId为空。	`org.activiti...ActivitiErrorEvent`
ACTIVITY_COMPENSATE	一个节点将要被补偿。事件包含 了将要执行补偿的节点id。	`org.activiti...ActivitiActivityEver`
VARIABLE_CREATED	创建了一个变量。事件包含变量名，变量值和对应的分支或任务（如果存在）。	`org.activiti...ActivitiVariableEver`
VARIABLE_UPDATED	更新了一个变量。事件包含变量 名，变量值和对应的分支或任务（如果存在）。	`org.activiti...ActivitiVariableEver`
VARIABLE_DELETED	删除了一个变量。事件包含变量名，变量值和对应的分支或任务（如果存在）	`org.activiti...ActivitiVariableEver`
TASK_ASSIGNED		`org.activiti...ActivitiEntityEvent`
TASK_CREATED	创建了新任务。它位于ENTITY_CREATE事件之后。当任务是由流程创建时， 这个事件会在TaskListener执行之前被执行。	`org.activiti...ActivitiEntityEvent`
TASK_COMPLETED	任务被完成了。它会在ENTITY_DELETE事件之前触发。当任务是流程一部分时，事件会在流程继续运行之前， 后续事件将是ACTIVITY_COMPLETE，对应着完成任务的节点。	`org.activiti...ActivitiEntityEvent`
PROCESS_COMPLETED	流程已结束。在最后一个节点 的ACTIVITY_COMPLETED 事件之后触发。 当流程到达的状态，没有任何后续连线时， 流程就会结束。	`org.activiti…ActivitiEntityEvent`
PROCESS_CANCELLED	流程已取消。在流程实例删除前从运行时触发。流程实例被 API RuntimeService.deleteProcessInstance调用	`org.activiti…ActivitiCancelledEvent`
MEMBERSHIP_CREATED	用户被添加到一个组里。事件包 含了用户和组的id。	`org.activiti...ActivitiMembershipEv`
MEMBERSHIP_DELETED	用户被从一个组中删除。事件包 含了用户和组的id。	`org.activiti...ActivitiMembershipEv`
MEMBERSHIPS_DELETED	所有成员被从一个组中删除。在 成员删除之前触发这个事件，所以他们都是可以访问的。 因为性能方面的考虑，不会为每个成员触发单独的MEMBERSHIP_DELETED事件。	

引擎内部所有 ENTITY_* 事件都是与实体相关的。下面的列表展示了实体事件与实体的对应关 系：

- ENTITY_CREATED, ENTITY_INITIALIZED, ENTITY_DELETED: Attachment, Comment, Deployment, Execution, Group, IdentityLink, Job, Model, ProcessDefinition, ProcessInstance, Task, User.
- ENTITY_UPDATED: Attachment, Deployment, Execution, Group, IdentityLink, Job, Model, ProcessDefinition,

ProcessInstance, Task, User.

- ENTITY_SUSPENDED, ENTITY_ACTIVATED: ProcessDefinition, ProcessInstance/Execution, Task.

Additional remarks 附加信息

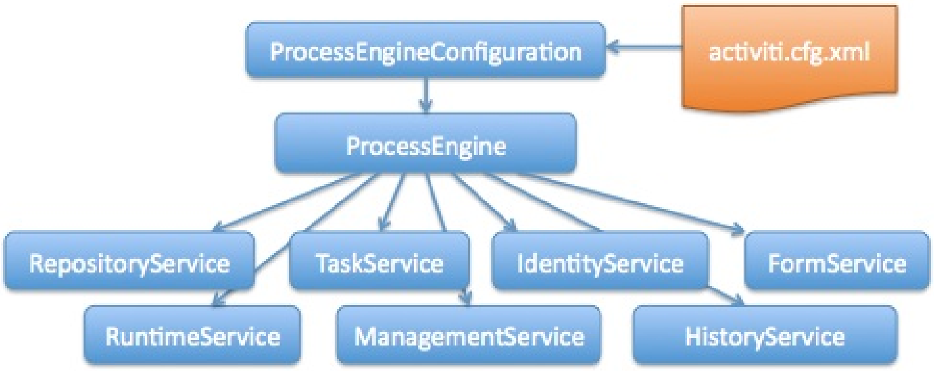
只有事件被发送，对应的引擎内的监听器才会被通知到。你有很多引擎 - 在同一个数据库运行 - 事件只会发送给注册到对应引擎的监听器。其他引擎发生的事件不会发送给这个监听器，无论实际上它们运行在同一个或不同的JVM中。

对应的事件类型（对应实体）都包含对应的实体。根据类型或事件，这些实体不能再进行更新（比如，当实例以被删除）。可能的话，使用事件提供的 EngineServices 来以安全的方式来操作引擎。即使如此，你需要小心的对事件对应的实体进行更新/操作。

没有对应历史的实体事件，因为它们都有运行阶段的对应实体。

The Process Engine API and services 流程引擎的 API 和服务

引擎 API 是与 Activiti 打交道的最常用方式。 我们从 ProcessEngine 开始， 创建它的很多种方法都已经在 配置章节中有所涉及。 从 ProcessEngine 中，你可以获得很多囊括工作流/BPM 方法的服务。 ProcessEngine 和服务类都是线程安全的。你可以在整个服务器中仅保持它们的一个引用就可以了。




```
ProcessEngine processEngine = ProcessEngines.getDefaultProcessEngine();

RuntimeService runtimeService = processEngine.getRuntimeService();
RepositoryService repositoryService = processEngine.getRepositoryService();
TaskService taskService = processEngine.getTaskService();
ManagementService managementService = processEngine.getManagementService();
IdentityService identityService = processEngine.getIdentityService();
HistoryService historyService = processEngine.getHistoryService();
FormService formService = processEngine.getFormService();
```

ProcessEngines.getDefaultProcessEngine() 会在第一次调用时 初始化并创建一个流程引擎，以后再调用就会返回相同的流程引擎。 使用对应的方法可以创建和关闭所有流程引擎：ProcessEngines.init() 和 ProcessEngines.destroy()。

ProcessEngines 会扫描所有 activiti.cfg.xml 和 activiti-context.xml 文件。 对于 activiti.cfg.xml 文件，流程引擎会使用Activiti 的经典方式构建：
ProcessEngineConfiguration.createProcessEngineConfigurationFromInputStream(inputStream).buildProcessEngine() 对于 activiti-context.xml 文件，流程引擎会使用 Spring 方法构建：先创建一个 Spring 的环境，然后通过环境获得流程引擎。

所有服务都是无状态的。这意味着可以在多节点集群环境下运行 ctiviti，每个节点都指向同一个数据库， 不用担心哪个机器实际执行前端的调用。 无论在哪里执行服务都没有问题。

RepositoryService 可能是使用 Activiti 引擎时最先接触的服务。 它提供了管理和控制发布包和流程定义的操作。 这里不涉及太多细节，流程定义是 BPMN 2.0 流程的 Java 实现。 它包含了一个流程每个环节的结构和行为。 发布包是 Activiti 引擎的打包单位。一个发布包可以包含多个 BPMN 2.0 xml 文件和其他资源。 开发者可以自由选择把任意资源包含到发布包中。 既可以把一个单独的 BPMN 2.0 xml 文件放到发布包里，也可以把整个流程和相关资源都放在一起。 （比如，'hr-processes' 实例可以包含hr流程相关的任何资源）。 可以通过 RepositoryService 来部署这种发布包。 发布一个发布包，意味着把它上传到引擎中，所有流 程都会在保存进数据库之前分析解析好。 从这点来说，系统知道这个发布包的存在，发布包中包含的流程就已经可以启动了。

除此之外，服务可以

- 查询引擎中的发布包和流程定义。
- 暂停或激活发布包，对应全部和特定流程定义。 暂停意味着它们不能再执行任何操作了，激活是对应的反向操作。
- 获得多种资源，像是包含在发布包里的文件，或引擎自动生成的流程图。
- 获得流程定义的pojo版本， 可以用来通过java解析流程，而不必通过xml。

正如 RepositoryService 负责静态信息（比如，不会改变的数据，至少是不怎么改变的），**RuntimeService** 正好是完全相反的。它负责启动一个流程定义的新实例。 如上所述，流程定义定义了流程各个节点的结构和行为。 流程实例就是这样一个流程定义的实例。对每个流程定义来说，同一时间会有很多实例在执行。 RuntimeService 也可以用来获取和保存流程变量。这些数据是特定于某个流程实例的，并会被很多流程中的节点使用 （比如，一个排他网关常常使用流程变量来决定选择哪条路径继续流程）。 Runtimeservice 也能查询流程实例和执行。 执行对应 BPMN 2.0 中的'token'。基本上执行指向流程实例当前在哪里。 最后，RuntimeService 可以在流程实例等待外部触发时使用，这时可以用来继续流程实例。 流程实例可以有很多暂停状态，而服务提供了多种方法来'触发'实例， 接受外部触发后，流程实例就会继续向下执行。

任务是由系统中真实人员执行的，它是 Activiti 这类 BPMN 引擎的核心功能之一。 所有与任务有关的功能都包含在**TaskService**中：

- 查询分配给用户或组的任务
- 创建独立运行任务。这些任务与流程实例无关。
- 手工设置任务的执行者，或者这些用户通过何种方式与任务关联。
- 认领并完成一个任务。认领意味着一个人期望成为任务的执行者， 即这个用户会完成这个任务。完成意味着“做这个任务要求的事情”。 通常来说会有很多种处理形式。

IdentityService 非常简单。它可以管理（创建，更新， 删除， 查询...）群组和用户。 请注意， Activiti 执行时并没有对用户进行检查。 例如，任务可以分配给任何人，但是引擎不会校验系统中是否存在这个用户。 这是 Activiti 引擎也可以使用外部服务，比如 ldap，活动目录，等等。

FormService 是一个可选服务。即使不使用它，Activiti 也可以完美运行，不会损失任何功能。这个服务提供了启动表单和任务表单两个概念。 启动表单会在流程实例启动之前展示给用户， 任务表单会在用户完成任务时展示。Activiti 支持在 BPMN 2.0 流程定义中设置这些表单。 这个服务以一种简单的方式将数据暴露出来。再次重申，它是可选的， 表单也不一定要嵌入到流程定义中。

HistoryService提供了 Activiti 引擎的所有历史数据。 在执行流程时，引擎会保存很多数据（根据配置），比如流程实例启动时间，任务的参与者， 完成任务的时间，每个流程实例的执行路径，等等。 这个服务主要通过查询功能来获得这些数据。

ManagementService在使用 Activiti 的定制环境中基本上不会用到。 它可以查询数据库的表和表的元数据。另外，它提供了查询和管理异步操作的功能。 Activiti 的异步操作用途很多，比如定时器，异步操作， 延迟暂停、激活，等等。后续，会讨论这些功能的更多细节。

可以从javadocs中获得这些服务和引擎 API 的更多信息。

Exception strategy 异常策略

Activiti 中的基础异常为org.activiti.engine.ActivitiException，一个非检查异常。 这个异常可以在任何时候被 API 抛出，不过特定方法抛出的"特定"的异常都记录在 javadocs中。 例如，下面的 TaskService：

```
/**
 * Called when the task is successfully executed.
 * @param taskId the id of the task to complete, cannot be null.
 * @throws ActivitiObjectNotFoundException when no task exists with the given id.
 */
 void complete(String taskId);
```

在上面的例子中，当传入一个不存在的任务的 id 时，就会抛出异常。 同时，javadoc 明确指出 **taskId** 不能为 **null**，如果传入 **null**， 就会抛出 **ActivitiIllegalArgumentException**。

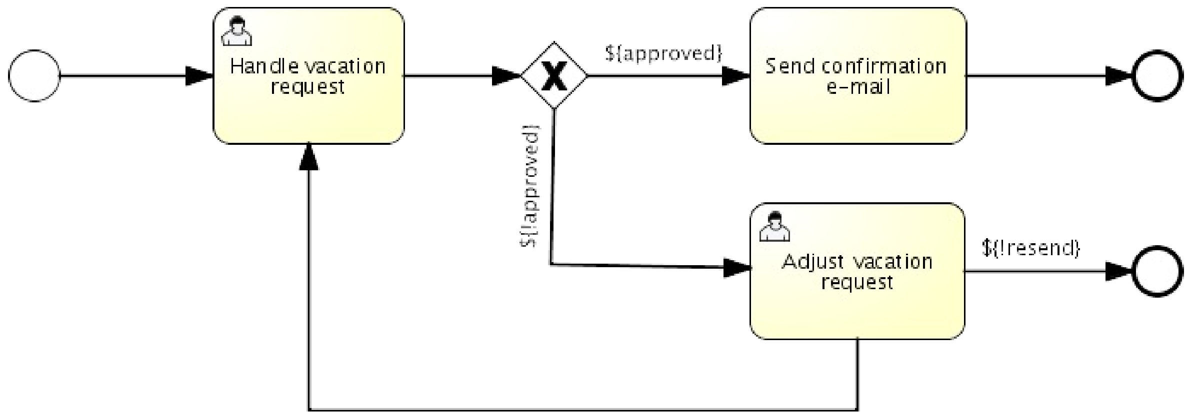
我们希望避免过多的异常继承，下面的子类用于特定的场合。 流程引擎和API 调用的其他场合不会使用下面的异常， 它们会抛出一个普通的ActivitiExceptions。

- ActivitiWrongDbException：当 Activiti 引擎发现数据库版本号和引擎版本号不一致时抛出。
- ActivitiOptimisticLockingException：对同一数据进行并发方法并出现乐观锁时抛出。
- ActivitiClassLoadingException：当无法找到需要加载的类或在加载类时出现了错误（比如，JavaDelegate，TaskListener等。)
- ActivitiObjectNotFoundException：当请求或操作的对应不存在时抛出。
- ActivitiIllegalArgumentException：这个异常表示调用Activiti API 时传入了一个非法的参数，可能是引擎配置中的非法值，或提供了一个非法制，或流程定义中使用的非法值
- ActivitiTaskAlreadyClaimedException：当任务已经被认领了，再调用 taskService.claim(...) 就会抛出。

Working with the Activiti services 使用 Activiti 服务

像上面介绍的那样，要想操作 Activiti 引擎，需要通过 org.activiti.engine.ProcessEngine 实例暴露的服务。 下面的代码假设你已经拥有了一个可以运行的 Activiti 环境。 你就可以操作一个org.activiti.engine.ProcessEngine。 如果只想简单尝试一下代码， 可以下载或者复制 Activiti单元测试模板 ， 导入到 IDE 中，把testUserguideCode() 方法添加到 org.activiti.MyUnitTest 中。

这个小例子的最终目标是做一个工作业务流程， 演示公司中简单的请假申请：



Deploying the process 部署流程

任何与“静态”资源有关的数据（比如流程定义）都可以通过 **RepositoryService**访问。 从概念上讲，所以静态数据都是Activiti的资源内容。

在src/test/resources/org/activiti/test 目录下创建一个新的 xml 文件 VacationRequest.bpmn20.xml（如果不使用单元测试模板，你也可以在任何地方创建）， 内容如下。注意这一章不会解释例子中使用的 xml 结构。 如果有需要可以先阅读bpmn 2.0章来了解这些。

```
<?xml version="1.0" encoding="UTF-8" ?>
<definitions id="definitions"
             targetNamespace="http://activiti.org/bpmn20"
             xmlns="http://www.omg.org/spec/BPMN/20100524/MODEL"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xmlns:activiti="http://activiti.org/bpmn">

  <process id="vacationRequest" name="Vacation request">

    <startEvent id="request" activiti:initiator="employeeName">
      <extensionElements>
        <activiti:formProperty id="numberOfDays" name="Number of days" type="long" value="1" required="true"/>
        <activiti:formProperty id="startDate" name="First day of holiday (dd-MM-yyy)" datePattern="dd-MM-yyyy hh:mm" ty
        <activiti:formProperty id="vacationMotivation" name="Motivation" type="string" />
      </extensionElements>
    </startEvent>
    <sequenceFlow id="flow1" sourceRef="request" targetRef="handleRequest" />

    <userTask id="handleRequest" name="Handle vacation request" >
      <documentation>
        ${employeeName} would like to take ${numberOfDays} day(s) of vacation (Motivation: ${vacationMotivation}).
      </documentation>
      <extensionElements>
         <activiti:formProperty id="vacationApproved" name="Do you approve this vacation" type="enum" required="true">
          <activiti:value id="true" name="Approve" />
          <activiti:value id="false" name="Reject" />
        </activiti:formProperty>
```

```
        <activiti:formProperty id="managerMotivation" name="Motivation" type="string" />
      </extensionElements>
      <potentialOwner>
        <resourceAssignmentExpression>
          <formalExpression>management</formalExpression>
        </resourceAssignmentExpression>
      </potentialOwner>
    </userTask>
    <sequenceFlow id="flow2" sourceRef="handleRequest" targetRef="requestApprovedDecision" />

    <exclusiveGateway id="requestApprovedDecision" name="Request approved?" />
    <sequenceFlow id="flow3" sourceRef="requestApprovedDecision" targetRef="sendApprovalMail">
      <conditionExpression xsi:type="tFormalExpression">${vacationApproved == 'true'}</conditionExpression>
    </sequenceFlow>

    <task id="sendApprovalMail" name="Send confirmation e-mail" />
    <sequenceFlow id="flow4" sourceRef="sendApprovalMail" targetRef="theEnd1" />
    <endEvent id="theEnd1" />

    <sequenceFlow id="flow5" sourceRef="requestApprovedDecision" targetRef="adjustVacationRequestTask">
      <conditionExpression xsi:type="tFormalExpression">${vacationApproved == 'false'}</conditionExpression>
    </sequenceFlow>

    <userTask id="adjustVacationRequestTask" name="Adjust vacation request">
      <documentation>
        Your manager has disapproved your vacation request for ${numberOfDays} days.
        Reason: ${managerMotivation}
      </documentation>
      <extensionElements>
        <activiti:formProperty id="numberOfDays" name="Number of days" value="${numberOfDays}" type="long" required="tr
        <activiti:formProperty id="startDate" name="First day of holiday (dd-MM-yyy)" value="${startDate}" datePattern=
        <activiti:formProperty id="vacationMotivation" name="Motivation" value="${vacationMotivation}" type="string" />
        <activiti:formProperty id="resendRequest" name="Resend vacation request to manager?" type="enum" required="true
          <activiti:value id="true" name="Yes" />
          <activiti:value id="false" name="No" />
        </activiti:formProperty>
      </extensionElements>
      <humanPerformer>
        <resourceAssignmentExpression>
          <formalExpression>${employeeName}</formalExpression>
        </resourceAssignmentExpression>
      </humanPerformer>
    </userTask>
    <sequenceFlow id="flow6" sourceRef="adjustVacationRequestTask" targetRef="resendRequestDecision" />

    <exclusiveGateway id="resendRequestDecision" name="Resend request?" />
    <sequenceFlow id="flow7" sourceRef="resendRequestDecision" targetRef="handleRequest">
      <conditionExpression xsi:type="tFormalExpression">${resendRequest == 'true'}</conditionExpression>
    </sequenceFlow>

     <sequenceFlow id="flow8" sourceRef="resendRequestDecision" targetRef="theEnd2">
      <conditionExpression xsi:type="tFormalExpression">${resendRequest == 'false'}</conditionExpression>
    </sequenceFlow>
    <endEvent id="theEnd2" />

  </process>

</definitions>
```

为了让 Activiti 引擎知道这个流程，我们必须先进行“部署”。 部署意味着引擎会把 BPMN 2.0 xml 解析成可以执行的东西，“部署包”中的所有流程定义都会添加到数据库中。 这样，当引擎重启时，它依然可以获得“已部署”的流程：

```
ProcessEngine processEngine = ProcessEngines.getDefaultProcessEngine();
RepositoryService repositoryService = processEngine.getRepositoryService();
repositoryService.createDeployment()
  .addClasspathResource("org/activiti/test/VacationRequest.bpmn20.xml")
  .deploy();

Log.info("Number of process definitions: " + repositoryService.createProcessDefinitionQuery().count());
```

可以阅读部署章节来了解更多关于部署的信息。

Starting a process instance 开始流程实例

把流程定义发布到 Activiti 引擎后，我们可以基于它发起新流程实例。 对每个流程定义，都可以有很多流程实例。 流程定义是“蓝图”，流程实例是它的一个运行的执行。

所有与流程运行状态相关的东西都可以通过 **RuntimeService** 获得。 有很多方法可以启动一个新流程实例。在下面的代码中，我们使用定义在流程定义 xml 中的 key 来启动流程实例。 我们也可以在流程实例启动时添加一些流程变量，因为第一个用户任务的表达式需要这些变量。 流程变量经常会被用到，因为它们赋予来自同一个流程定义的不同流程实例的特别含义。简单来说，流程变量是区分流程实例的关键。

```
Map<String, Object> variables = new HashMap<String, Object>();
variables.put("employeeName", "Kermit");
variables.put("numberOfDays", new Integer(4));
variables.put("vacationMotivation", "I'm really tired!");

RuntimeService runtimeService = processEngine.getRuntimeService();
ProcessInstance processInstance = runtimeService.startProcessInstanceByKey("vacationRequest", variables);

// Verify that we started a new process instance
Log.info("Number of process instances: " + runtimeService.createProcessInstanceQuery().count());
```

Completing tasks 完成任务

流程启动后，第一步就是用户任务。这是必须由系统用户处理的一个环节。 通常，用户会有一个“任务列表”，展示了所有须由整个用户处理的任务。 下面的代码展示了对应的查询可能是怎样的：

```
// Fetch all tasks for the management group
TaskService taskService = processEngine.getTaskService();
List<Task> tasks = taskService.createTaskQuery().taskCandidateGroup("management").list();
for (Task task : tasks) {
  Log.info("Task available: " + task.getName());
}
```

为了让流程实例继续运行，我们需要完成整个任务。对 Activiti 来说，就是需要 complete 任务。 下面的代码展示了如何做这件事：

```
Task task = tasks.get(0);

Map<String, Object> taskVariables = new HashMap<String, Object>();
taskVariables.put("vacationApproved", "false");
taskVariables.put("managerMotivation", "We have a tight deadline!");
taskService.complete(task.getId(), taskVariables);
```

流程实例会进入到下一个环节。在这里例子中， 下一环节允许员工通过表单调整原始的请假申请。员工可以重新提交请假申请， 这会使流程重新进入到第一个任务。

Suspending and activating a process 挂起，激活一个流程

我们可以挂起一个流程定义。当挂起流程定时时， 就不能创建新流程了（会抛出一个异常）。 可以通过 RepositoryService 挂起一个流程

```
repositoryService.suspendProcessDefinitionByKey("vacationRequest");
try {
  runtimeService.startProcessInstanceByKey("vacationRequest");
} catch (ActivitiException e) {
  e.printStackTrace();
}
```

要想重新激活一个流程定义，可以调用repositoryService.activateProcessDefinitionXXX 方法。也可以挂起一个流程实例。挂起时，流程不能继续执行（比如，完成任务会抛出异常）， 异步操作（比如定时器）也不会执行。挂起流程实例可以调用 runtimeService.suspendProcessInstance 方法。 激活流程实例可以调用 runtimeService.activateProcessInstanceXXX 方法。

Further reading 扩展阅读

上面章节中我们仅仅覆盖了 Activiti 功能的表层。 未来我们会继续扩展这些章节，以覆盖更多 Activiti API。 当然，像其他开源项目一样，学习的最好方式 是研究代码，阅读 Javadocs。

Query API 查询 API

有两种方法可以从引擎中查询数据：查询 API 和原生查询。查询 API 提供了完全类型安全的 API。 你可以为自己的查询条件添加很多条件 （所有条件都以 AND 组合）和精确的排序条件。下面的代码展示了一个例子：

```
List<Task> tasks = taskService.createTaskQuery()
         .taskAssignee("kermit")
         .processVariableValueEquals("orderId", "0815")
         .orderByDueDate().asc()
         .list();
```

有时，你需要更强大的查询，比如使用 OR 条件或不能使用查询 API 实现的条件。 这时，我们推荐原生查询，它让你可以编写自己的SQL查询。 返回类型由你使用的查询对象决定，数据会映射到正确的对象上。比如，任务，流程实例，执行，等等。 因为查询会作用在数据库上，你必须使用数据库中定义的表名和列名；这要求了解内部数据结构， 因此使用原生查询时一定要注意。表名可以通过 API 获得，可以尽量减少对数据库的依赖。

```
List<Task> tasks = taskService.createNativeTaskQuery()
    .sql("SELECT count(*) FROM " + managementService.getTableName(Task.class) + " T WHERE T.NAME_ = #{taskName}")
    .parameter("taskName", "gonzoTask")
    .list();

long count = taskService.createNativeTaskQuery()
    .sql("SELECT count(*) FROM " + managementService.getTableName(Task.class) + " T1, "
           + managementService.getTableName(VariableInstanceEntity.class) + " V1 WHERE V1.TASK_ID_ = T1.ID_")
    .count();
```

Variables 变量

每个流程实例需要并且使用数据来执行存在的步骤。在 Activiti,这些数据称为 variables（变量）,并存储在数据库中。变量可用于表达式(例如在单独的网关中选择正确的流出序列流),在 java 服务任务调用外部服务(例如提供输入或存储服务调用的结果),等等。

一个流程实例可以有变量(称为 process variables 流程变量),但也可以执行(流程是活动的特定指针)并且用户任务可以有变量。一个流程实例可以拥有任意数量的变量。每个变量存储在 ACT_RU_VARIABLE 数据库表中的一行。

任何 startProcessInstanceXXX 方法都有一个可选的参数来提供变量，当流程实例创建和开始时。例如, RuntimeService:

```
ProcessInstance startProcessInstanceByKey(String processDefinitionKey, Map<String, Object> variables);
```

在流程执行时可以添加变量。例如(RuntimeService):

```
void setVariable(String executionId, String variableName, Object value);
void setVariableLocal(String executionId, String variableName, Object value);
void setVariables(String executionId, Map<String, ? extends Object> variables);
void setVariablesLocal(String executionId, Map<String, ? extends Object> variables);
```

注意变量可以设置在本地， 对于一个给定的执行(记住一个流程实例由一个执行树组成)。变量只在执行时可见,并且不会高于执行树。当数据不应该传播到流程实例级别,或变量有在流程实例中特定路径的新值(例如当使用并行路径时)，这可能是有用的。

变量也可以再次获取,如下所示。注意,类似的方法在 TaskService 存在。这意味着任务跟执行一样,可以使用局部变量,为了任务的持续时间 而 alive (存活)。

```
Map<String, Object> getVariables(String executionId);
Map<String, Object> getVariablesLocal(String executionId);
Map<String, Object> getVariables(String executionId, Collection<String> variableNames);
Map<String, Object> getVariablesLocal(String executionId, Collection<String> variableNames);
Object getVariable(String executionId, String variableName);
<T> T getVariable(String executionId, String variableName, Class<T> variableClass);
```

变量经常使用在 Java delegates, 表达式, execution- 或者 tasklisteners ,脚本,等等。在这些结构,当前执行或任务对象是可用的,它可以用于变量设置和/或检索。最简单的方法是:

```
execution.getVariables();
execution.getVariables(Collection<String> variableNames);
execution.getVariable(String variableName);

execution.setVariables(Map<String, object> variables);
execution.setVariable(String variableName, Object value);
```

注意,本地变体也可用于上述所有。

历史(和向后兼容的原因),在上面的任何调用,实际上在幕后所有变量将从数据库中获取。这意味着,如果你有10个变量,并通过 getVariable("myVariable") 只有一次,在幕后其他9个将获取和缓存。这不是坏事,因为后续调用不会再接触到数据库。例如,当您的流程定义有三个连续的服务任务(因此有一个数据库事务),使用一个调用来获取所有变量在第一个服务任务的时候，这样可能比在每个服务任务分别获取所需的变量要好。注意,这个应用在获取和设置变量时。

当然,当使用大量的变量或者只是当您想要严格控制数据库查询和交互的时候,这是不合适的。Activiti 5.17 以来,新方法介绍了

给一个更严格的控制,通过添加一个可选参数的新方法,告诉引擎是否需要在幕后将所有变量获取并缓存:

```
Map<String, Object> getVariables(Collection<String> variableNames, boolean fetchAllVariables);
Object getVariable(String variableName, boolean fetchAllVariables);
void setVariable(String variableName, Object value, boolean fetchAllVariables);
```

当参数 fetchAllVariables 为 true 时,上述行为将完全一样:当获取或设置一个变量,所有其他变量将获取和缓存。

然而,当值是 false 时,将使用特定的查询，其他变量将不会获取和缓存。只有当前问题的变量的值将缓存为后续使用。

Expressions 表达式

Activiti 使用 UEL 处理表达式。UEL 即 Unified Expression Language (统一表达式语言)，它是 EE6 规范的一部分（参考 EE6规范）。为了在所有运行环境都支持最新 UEL 的所有功能，我们使用了一个 JUEL 的修改版本。

表达式可以用在很多场景下，比如 Java 服务任务，执行监听器，任务监听器和条件流。 虽然有两重表达式，值表达式和方法表达式，Activiti进行了抽象，所以两者可以同样使用在需 要表达式的场景中。

- Value expression(值表达式)：解析为值。默认，所有流程变量都可以使用。所有 spring bean（spring环境中）也可以使用在表达式中。 一些实例：

  ${myVar} ${myBean.myProperty}

- Method expression(方法表达式)：调用一个方法，使用或不使用参数。当调用一个无参数的方法时，记得在方法名后添加空的括号（以区分值表达式）。 传递的参数可以是字符串也可以是表达式，它们会被自动解析。例子：

  ${printer.print()} ${myBean.addNewOrder('orderName')} ${myBean.doSomething(myVar, execution)}

注意这些表达式支持解析原始类型（包括比较），bean，list，数组和map。

在所有流程实例中，表达式中还可以使用一些默认对象：

- execution：DelegateExecution 提供外出执行的额外信息。
- task：DelegateTask 提供当前任务的额外信息。注意，只对任务监听器的表达式有效。
- authenticatedUserId：当前登录的用户id。如果没有用户登录，这个变量就不可用。

想要更多具体的使用方式和例子，参考 spring 中的表达式，Java 服务任务，执行监听器，任务监听器和条件流。

Unit testing 单元测试

业务流程是软件项目的一部分，它也应该和普通的业务流程一样进行测试： 使用单元测试。因为 Activiti 是一个嵌入式的 java 引擎，为业务流程编写单元测试和写普通单元测试完全一样。

Activiti 支持 JUnit 3和4进行单元测试。使用 JUnit 3时， 必须集成 org.activiti.engine.test.ActivitiTestCase。它通过保护的成员变量提供 ProcessEngine 和服务，在测试的 setup() 中， 默认会使用classpath 下的 activiti.cfg.xml 初始化流程引擎。 想使用不同的配置文件，可以重写 getConfigurationResource() 方法。 如果配置文件相同的话，对应的流程引擎会被静态缓存， 就可以用于多个单元测试。

继承了 ActivitiTestCase,你可以在测试方法上使用 org.activiti.engine.test.Deployment 注解。测试执行前，与测试类在同一个包下的， 格式为 testClassName.testMethod.bpmn20.xml的资源文件，会被部署。 测试结束后，发布包也会被删除，包括所有相关的流程实例，任务，等等。Deployment 注解也可以直接设置资源的位置。 参考Javadocs 获得更多信息。

把这些放在一起，JUnit 3 测试看起来像这样。

```
public class MyBusinessProcessTest extends ActivitiTestCase {

  @Deployment
  public void testSimpleProcess() {
    runtimeService.startProcessInstanceByKey("simpleProcess");

    Task task = taskService.createTaskQuery().singleResult();
    assertEquals("My Task", task.getName());

    taskService.complete(task.getId());
    assertEquals(0, runtimeService.createProcessInstanceQuery().count());
  }
}
```

要想在使用 JUnit 4 编写单元测试时获得同样的功能， 可以使 用 org.activiti.engine.test.ActivitiRule。 通过它，可以通过 getter 方法获得流程引擎和各种服务。 和 ActivitiTestCase 一样（参考上面章节）， 使用这个 Rule 也会启用 org.activiti.engine.test.Deployment 注解（参考上面章节使用和配置的介绍）， 它会在 classpath 下查找默认的配置文件。 如果配置文件相同的话，对应的流程引擎会被静态缓存， 就可以用于多个单元测试。

下面的代码演示了 JUnit 4 单元测试并使用了 ActivitiRule 的例子。

```
public class MyBusinessProcessTest {

  @Rule
  public ActivitiRule activitiRule = new ActivitiRule();

  @Test
  @Deployment
  public void ruleUsageExample() {
    RuntimeService runtimeService = activitiRule.getRuntimeService();
    runtimeService.startProcessInstanceByKey("ruleUsage");

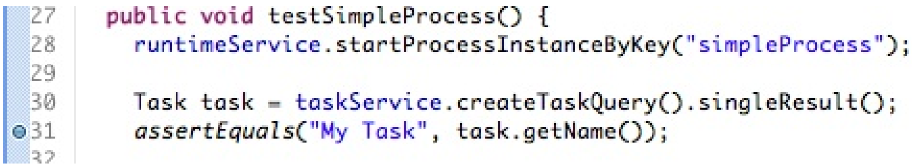
    TaskService taskService = activitiRule.getTaskService();
    Task task = taskService.createTaskQuery().singleResult();
    assertEquals("My Task", task.getName());

    taskService.complete(task.getId());
    assertEquals(0, runtimeService.createProcessInstanceQuery().count());
  }
}
```

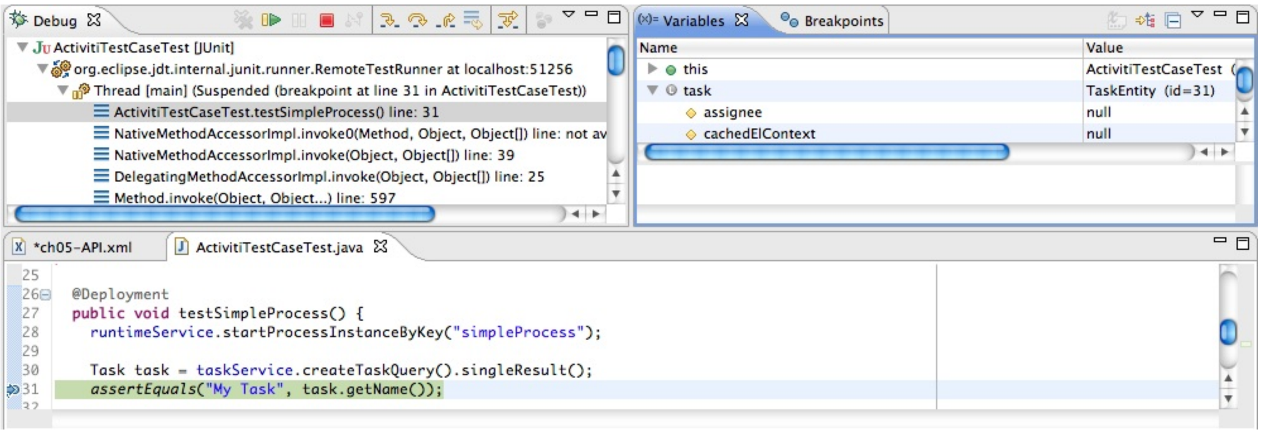
Debugging unit tests 调试单元测试

当使用内存数据库 H2 进行单元测试时，下面的教程会告诉我们 如何在调试环境下更容易的监视 Activiti 的数据库。 这里的截图都是基于eclipse，这种机制很容易复用到其他 IDE 下。

假设我们已经在单元测试里设置了一个断点。 Eclipse 里，在代码左侧双击：

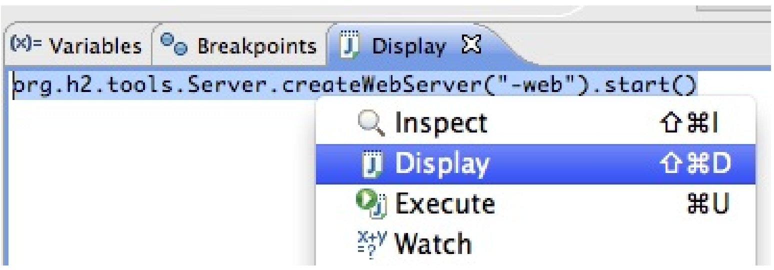


现在用调试模式运行单元测试（右击单元测试， 选择"运行为"和"单元测试"）， 测试会停在我们的断点上， 然后我们就可以监视测试的变量，它们显示在右侧面板里。



要监视Activiti的数据，打开"显示"窗口 （如果找不到，打开"窗口"->"显示视图"->"其他"，选择显示。） 并点击（代码已完成）

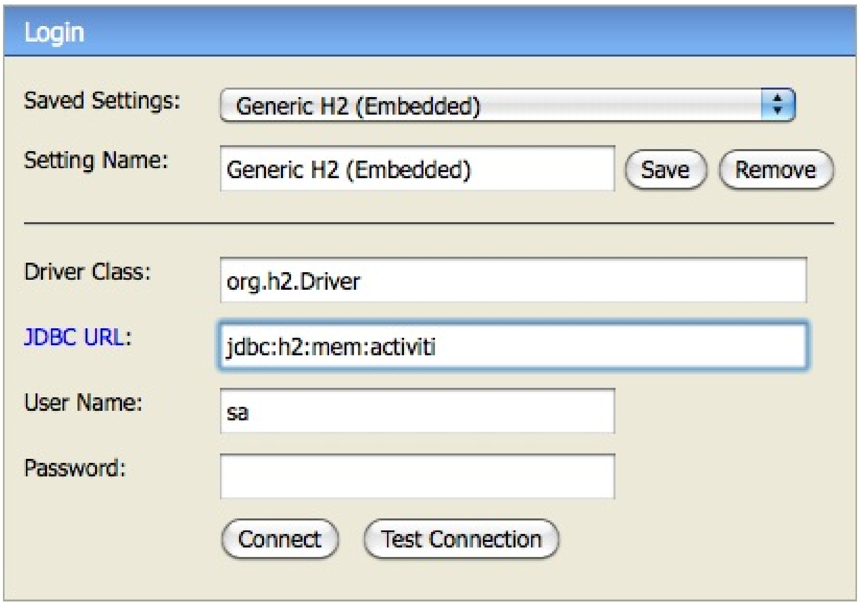
org.h2.tools.Server.createWebServer("-web").start()



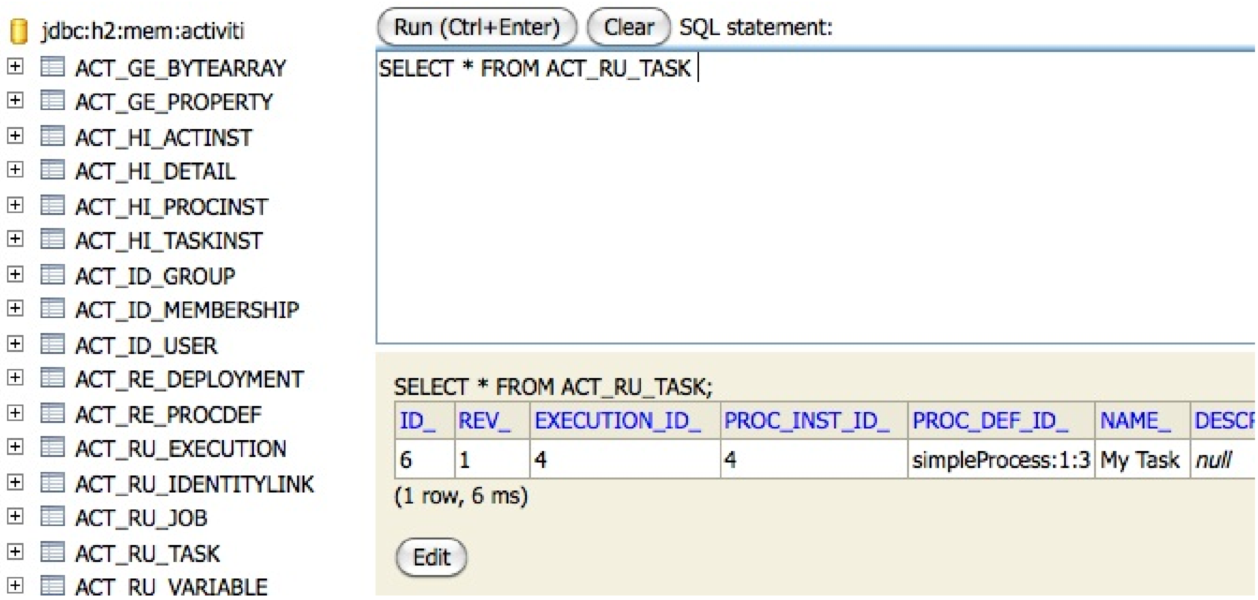
选择你点击的行，右击。然后选择"显示"（或者直接快捷方式就不用右击了）



现在打开一个浏览器，打开 http://localhost:8082， 输入内存数据库的 JDBC URL（默认为 jdbc:h2:mem:activiti）， 点击连接按钮。



你现在可以看到 Activiti 的数据，通过它们可以了解单元测试时如何以及为什么这样运行的。



The process engine in a web application 在 web 应用中的流程引擎

ProcessEngine 是线程安全的， 可以在多线程下共享。在 web 应用中， 意味着可以在容器启动时创建流程引擎， 在容器关闭时关闭流程引擎。 下面代码演示了如何编写一个 ServletContextListener 在普通的Servlet 环境下初始化和销毁流程引擎：

```
public class ProcessEnginesServletContextListener implements ServletContextListener {

  public void contextInitialized(ServletContextEvent servletContextEvent) {
    ProcessEngines.init();
  }

  public void contextDestroyed(ServletContextEvent servletContextEvent) {
    ProcessEngines.destroy();
  }

}
```

contextInitialized 方法会执行 ProcessEngines.init()。 这会查找 classpath 下的 activiti.cfg.xml 文 件， 根据配置文件创建一个 ProcessEngine（比如，多个 jar 中都包含配置文件）。 如果 classpath 中包含多个配置文件，确认它们有不同的名字。当需要使用流程引擎时，可以通过

```
ProcessEngines.getDefaultProcessEngine()
```

或者

```
ProcessEngines.getProcessEngine("myName");
```

当然，也可以使用其他方式创建流程引擎，可以参考配置章节中的描述。

ContextListener 中的 contextDestroyed 方法会执行ProcessEngines.destroy()， 这会关闭所有初始化的流程引擎。

Chapter 5. Spring integration 集成 Spring

虽然没有 Spring 你也可以使用 Activiti，但是我们提供了一些非常不错的集成特性。这一章我们将介绍这些特性。

ProcessEngineFactoryBean

可以把 ProcessEngine 作为一个普通的 Spring bean 进行配置。 类 org.activiti.spring.ProcessEngineFactoryBean 是集成的切入点。 这个 bean 需要一个流程引擎配置来创建流程引擎。这也意味着在文档的配置这一章的介绍属性的创建和配置对于 Spring 来说也是一样的。对于 Spring 集成的配置和流程引擎 bean 看起来像这样：

```
<bean id="processEngineConfiguration" class="org.activiti.spring.SpringProcessEngineConfiguration">
    ...
</bean>

<bean id="processEngine" class="org.activiti.spring.ProcessEngineFactoryBean">
  <property name="processEngineConfiguration" ref="processEngineConfiguration" />
</bean>
```

注意现在 processEngineConfiguration 的 bean 是使用 org.activiti.spring.SpringProcessEngineConfiguration 类。

Transactions 事务

我们将会一步一步地解释在 Spring examples 中公布的 SpringTransactionIntegrationTest 下面是我们使用这个例子的Spring配置文件（你可以在SpringTransactionIntegrationTestcontext.xml找到它）以下展示的部分包括数据源（dataSource）， 事务管理器（transactionManager），流程引擎（processEngine）和 Activiti 引擎服务。

当把数据源（DataSource）传递给 SpringProcessEngineConfiguration （使用"dataSource"属性） 之后，Activiti内部使用了一个org.springframework.jdbc.datasource.TransactionAwareDataSourceProxy 代理来封装传递进来的数据源（DataSource）。这样做是为了确保从数据源（DataSource）获取的SQL连接能够与 Spring 的事物结合在一起发挥得更出色。这意味它不再需要在你的Spring配置中代理数据源（dataSource）了。 然而它仍然允许你传递一个TransactionAwareDataSourceProxy 到SpringProcessEngineConfiguration中。在这个例子中并不会发生多余的包装。

为了确保在你的 **Spring** 配置中申明的一个**TransactionAwareDataSourceProxy**，你不能把使用它的应用交给**Spring**事物控制的资源。（例如 **DataSourceTransactionManager** 和 **JPATransactionManager** 需要非代理的数据源 ）

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:context="http://www.springframework.org/schema/context"
       xmlns:tx="http://www.springframework.org/schema/tx"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans   http://www.springframework.org/schema/beans/sp
                           http://www.springframework.org/schema/context http://www.springframework.org/schema/context/
                           http://www.springframework.org/schema/tx      http://www.springframework.org/schema/tx/sprin

  <bean id="dataSource" class="org.springframework.jdbc.datasource.SimpleDriverDataSource">
    <property name="driverClass" value="org.h2.Driver" />
    <property name="url" value="jdbc:h2:mem:activiti;DB_CLOSE_DELAY=1000" />
    <property name="username" value="sa" />
    <property name="password" value="" />
  </bean>

  <bean id="transactionManager" class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
    <property name="dataSource" ref="dataSource" />
  </bean>

  <bean id="processEngineConfiguration" class="org.activiti.spring.SpringProcessEngineConfiguration">
    <property name="dataSource" ref="dataSource" />
    <property name="transactionManager" ref="transactionManager" />
    <property name="databaseSchemaUpdate" value="true" />
    <property name="jobExecutorActivate" value="false" />
  </bean>

  <bean id="processEngine" class="org.activiti.spring.ProcessEngineFactoryBean">
    <property name="processEngineConfiguration" ref="processEngineConfiguration" />
  </bean>

  <bean id="repositoryService" factory-bean="processEngine" factory-method="getRepositoryService" />
  <bean id="runtimeService" factory-bean="processEngine" factory-method="getRuntimeService" />
  <bean id="taskService" factory-bean="processEngine" factory-method="getTaskService" />
  <bean id="historyService" factory-bean="processEngine" factory-method="getHistoryService" />
  <bean id="managementService" factory-bean="processEngine" factory-method="getManagementService" />

...
```

Spring 配置文件的其余部分包含 beans 和我们将要在这个特有的例子中的配置：

```
<beans>
  ...
  <tx:annotation-driven transaction-manager="transactionManager"/>

  <bean id="userBean" class="org.activiti.spring.test.UserBean">
    <property name="runtimeService" ref="runtimeService" />
  </bean>
```

```
  <bean id="printer" class="org.activiti.spring.test.Printer" />

</beans>
```

首先使用任意的一种 Spring 创建应用上下文的方式创建其 Spring 应用上下文。在这个例子中你可以使用类路径下面的 XML 资源来配置我们的Spring 应用上下文：

```
ClassPathXmlApplicationContext applicationContext =
    new ClassPathXmlApplicationContext("org/activiti/examples/spring/SpringTransactionIntegrationTest-context.xml");
```

或者, 如果它是一个测试的话:

```
@ContextConfiguration("classpath:org/activiti/spring/test/transaction/SpringTransactionIntegrationTest-context.xml")
```

然后我们就可以得到 Activiti 的服务 beans 并且调用该服务上面的方 法。ProcessEngineFactoryBean 将会对该服务添加一些额外的拦截器，在 Activiti 服务上面的方法使用的是 Propagation.REQUIRED 事物语义。所以，我们可以使用 repositoryService 去部署一个流程，如下所示：

```
RepositoryService repositoryService = (RepositoryService) applicationContext.getBean("repositoryService");
String deploymentId = repositoryService
  .createDeployment()
  .addClasspathResource("org/activiti/spring/test/hello.bpmn20.xml")
  .deploy()
  .getId();
```

其他相同的服务也是同样可以这么使用。在这个例子中，Spring 的事物将会围绕在 userBean.hello() 上， 并且调用 Activiti 服务的方法也会加入到这个事物中。

```
UserBean userBean = (UserBean) applicationContext.getBean("userBean");
userBean.hello();
```

这个 UserBean 看起来像这样。记得在上面 Spring bean 的配置中我们把 repositoryService 注入到 userBean 中。

```
public class UserBean {

  /** injected by Spring */
  private RuntimeService runtimeService;

  @Transactional
  public void hello() {
    // here you can do transactional stuff in your domain model
    // and it will be combined in the same transaction as
    // the startProcessInstanceByKey to the Activiti RuntimeService
    runtimeService.startProcessInstanceByKey("helloProcess");
  }

  public void setRuntimeService(RuntimeService runtimeService) {
    this.runtimeService = runtimeService;
  }
}
```

Expressions 表达式

当使用 ProcessEngineFactoryBean 时候，默认情况下，在 BPMN 流程中的所有表达式都将会'看见'所有的 Spring beans。它可以限制你在表达式中暴露出的 beans 或者甚至可以在你的配置中使用一个 Map 不暴露任何 beans。下面的例子暴露了一个单例 bean（printer），可以把 "printer" 当作关键字使用。想要不暴露任何**beans**，仅仅只需要在 **SpringProcessEngineConfiguration** 中传递一个空的 **list** 作为**'beans'**的属性。当不设置**'beans'**的属性时，在应用上下文中 **Spring beans** 都是可以使用的。

```
<bean id="processEngineConfiguration" class="org.activiti.spring.SpringProcessEngineConfiguration">
  ...
  <property name="beans">
    <map>
      <entry key="printer" value-ref="printer" />
    </map>
  </property>
</bean>

  <bean id="printer" class="org.activiti.examples.spring.Printer" />
```

现在暴露出来的 beans 就可以在表达式中使用：例如，在 SpringTransactionIntegrationTest 中的 hello.bpmn20.xml 展示的是如何使用 UEL 方法表达式去调用 Spring bean 的方法：

```
<definitions id="definitions" ...>

  <process id="helloProcess">

    <startEvent id="start" />
    <sequenceFlow id="flow1" sourceRef="start" targetRef="print" />

    <serviceTask id="print" activiti:expression="#{printer.printMessage()}" />
    <sequenceFlow id="flow2" sourceRef="print" targetRef="end" />

    <endEvent id="end" />

  </process>

</definitions>
```

这里的 Printer 看起来像这样：

```
public class Printer {

  public void printMessage() {
    System.out.println("hello world");
  }
}
```

并且 Spring bean 的配置（如上文所示）看起来像这样：

```
<beans ...>
  ...

  <bean id="printer" class="org.activiti.examples.spring.Printer" />

</beans>
```

Automatic resource deployment 资源自动部署

Spring 的集成也有一个专门用于对资源部署的特性。在流程引擎的配置中，你可以指定一组资源。当流程引擎被创建的时候， 所有在这里的资源都将会被自动扫描与部署。在这里有过滤以防止资源重新部署，只有当这个资源真正发生改变的时候，它才会向 Activiti 使用的数据库创建新的部署。 这对于很多用例来说，当 Spring 容器经常重启的情况下（例如测试），使用它是非常不错的选择。

这里有一个例子：

```
<bean id="processEngineConfiguration" class="org.activiti.spring.SpringProcessEngineConfiguration">
  ...
  <property name="deploymentResources" value="classpath*:/org/activiti/spring/test/autodeployment/autodeploy.*.bpmn20.x
</bean>

<bean id="processEngine" class="org.activiti.spring.ProcessEngineFactoryBean">
  <property name="processEngineConfiguration" ref="processEngineConfiguration" />
</bean>
```

默认，上面的配置会把所有匹配的资源发布到 Activiti 引擎的一个单独发布包下。用来检测防止未修改资源重复发布的机制会作用到整个发布包中。 有时候，这可能不是你想要的。比如，如果你发布了很多流程资源，但是只修改里其中某一个单独的流程定义， 整个发布包都会被认为变更了，导致整个发布包下的所有流程定义都会被重新发布， 结果就是每个流程定义都生成了新版本，虽然其中只有一个流程发生了改变。

为了定制发布方式，你可以为 SpringProcessEngineConfiguration 指定一个额外的参数 deploymentMode。 这个参数指定了匹配多个资源时的发布处理方式。默认下这个参数支持设置三个值：

- default: 把所有资源放在一个单独的发布包中，对这个发布包进行重复检测。 这是默认值，如果你没有指定参数值，就会使用它。
- single-resource: 为每个单独的资源创建一个发布包，并对这些发布包进行重复检测。 你可以单独发布每个流程定义，并在修改流程定义后只创建一个新的流程定义版本。
- resource-parent-folder: 把放在同一个上级目录下的资源发布在一个单独的发布包中，并对发布包进行重复检测。 当需要多资源需要创建发布包，但是需要根据共同的文件夹来组合一些资源时，可以使用它。

这儿有一个例子来演示将 deploymentMode 参数配置为 single-resource 的情况：

```
<bean id="processEngineConfiguration" class="org.activiti.spring.SpringProcessEngineConfiguration">
  ...
  <property name="deploymentResources" value="classpath*:/activiti/*.bpmn" />
  <property name="deploymentMode" value="single-resource" />
</bean>
```

如果想使用上面三个值之外的参数值，你需要自定义处理发布包的行为。 你可以创建一个 SpringProcessEngineConfiguration 的子类，重写getAutoDeploymentStrategy(String deploymentMode) 方 法。 这个方法中处理了对应 deploymentMode 的发布策略

Unit testing 单元测试

当集成 Spring 时，使用标准的 Activiti 测试工具类是非常容易的对业务流程进行测试。 下面的例子展示了如何在一个典型的基于Spring单元测试测试业务流程：

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration("classpath:org/activiti/spring/test/junit4/springTypicalUsageTest-context.xml")
public class MyBusinessProcessTest {

  @Autowired
  private RuntimeService runtimeService;

  @Autowired
  private TaskService taskService;

  @Autowired
  @Rule
  public ActivitiRule activitiSpringRule;

  @Test
  @Deployment
  public void simpleProcessTest() {
    runtimeService.startProcessInstanceByKey("simpleProcess");
    Task task = taskService.createTaskQuery().singleResult();
    assertEquals("My Task", task.getName());

    taskService.complete(task.getId());
    assertEquals(0, runtimeService.createProcessInstanceQuery().count());

  }
}
```

注意对于这种方式，你需要在 Spring 配置中（在上文的例子中它是自动注入的）定义一个 org.activiti.engine.test.ActivitiRulebean

```
<bean id="activitiRule" class="org.activiti.engine.test.ActivitiRule">
  <property name="processEngine" ref="processEngine" />
</bean>
```

Annotation-based configuration 基于注解的配置

[试验] @EnableActiviti注解相对较新，未来可能会有变更。

除了基于 XML 的配置以外，还可以选择基于注解的方式来配置 Spring 环境。 这与使用 XML 的方法非常相似，除了要使用 @Bean 注解， 而且配置是使用 java 编写的。 它已经可以直接用于 Activiti-Spring 的集成了：

首先介绍（需要 Spring 3.0+ ）的是 @EnableActiviti 注解。 最简单的用法如下所示：

```
@Configuration
  @EnableActiviti
  public static class SimplestConfiguration {

  }
```

它会创建一个 Spring 环境，并对 Activiti 流程引擎进行如下配置

- 默认的内存 H2 数据库，启用数据库自动升级。
- 一个简单的 DataSourceTransactionManager
- 一个默认的 SpringJobExecutor
- 自动扫描 processes/ 目录下的 bpmn20.xml 文件。

在这样一个环境里，可以直接通过注入操作 Activiti 引擎：

```
  @Autowired
  private ProcessEngine processEngine;

  @Autowired
  private RuntimeService runtimeService;

  @Autowired
  private TaskService taskService;

  @Autowired
  private HistoryService historyService;

  @Autowired
  private RepositoryService repositoryService;

  @Autowired
  private ManagementService managementService;

  @Autowired
  private FormService formService;
```

当然，默认值都可以自定义。比如，如果配置了 DataSource，它就会代替默认创建的数据库配置。 事务管理器，job 执行器和其他组件都与之相同。 比如如下配置：

```
  @Configuration
  @EnableActiviti
  public static class Config {

    @Bean
    public DataSource dataSource() {
        BasicDataSource basicDataSource = new BasicDataSource();
        basicDataSource.setUsername("sa");
        basicDataSource.setUrl("jdbc:h2:mem:anotherDatabase");
        basicDataSource.setDefaultAutoCommit(false);
        basicDataSource.setDriverClassName(org.h2.Driver.class.getName());
        basicDataSource.setPassword("");
```

```
        return basicDataSource;
    }

  }
```

其他数据库会代替默认的。下面介绍了更加复杂的配置。注意AbstractActivitiConfigurer 用法， 它暴露了流程引擎的 配置，可以用来对它的细节进行详细的配置。

```
@Configuration
@EnableActiviti
@EnableTransactionManagement(proxyTargetClass = true)
class JPAConfiguration {

    @Bean
    public OpenJpaVendorAdapter openJpaVendorAdapter() {
        OpenJpaVendorAdapter openJpaVendorAdapter = new OpenJpaVendorAdapter();
        openJpaVendorAdapter.setDatabasePlatform(H2Dictionary.class.getName());
        return openJpaVendorAdapter;
    }

    @Bean
    public DataSource dataSource() {
        BasicDataSource basicDataSource = new BasicDataSource();
        basicDataSource.setUsername("sa");
        basicDataSource.setUrl("jdbc:h2:mem:activiti");
        basicDataSource.setDefaultAutoCommit(false);
        basicDataSource.setDriverClassName(org.h2.Driver.class.getName());
        basicDataSource.setPassword("");
        return basicDataSource;
    }

    @Bean
    public LocalContainerEntityManagerFactoryBean entityManagerFactoryBean(
        OpenJpaVendorAdapter openJpaVendorAdapter, DataSource ds) {
        LocalContainerEntityManagerFactoryBean emf = new LocalContainerEntityManagerFactoryBean();
        emf.setPersistenceXmlLocation("classpath:/org/activiti/spring/test/jpa/custom-persistence.xml");
        emf.setJpaVendorAdapter(openJpaVendorAdapter);
        emf.setDataSource(ds);
        return emf;
    }

    @Bean
    public PlatformTransactionManager jpaTransactionManager(
        EntityManagerFactory entityManagerFactory) {
        return new JpaTransactionManager(entityManagerFactory);
    }

    @Bean
    public AbstractActivitiConfigurer abstractActivitiConfigurer(
        final EntityManagerFactory emf,
        final PlatformTransactionManager transactionManager) {

        return new AbstractActivitiConfigurer() {

            @Override
            public void postProcessSpringProcessEngineConfiguration(SpringProcessEngineConfiguration engine) {
                engine.setTransactionManager(transactionManager);
                engine.setJpaEntityManagerFactory(emf);
                engine.setJpaHandleTransaction(false);
                engine.setJobExecutorActivate(false);
                engine.setJpaCloseEntityManager(false);
                engine.setDatabaseSchemaUpdate(ProcessEngineConfiguration.DB_SCHEMA_UPDATE_TRUE);
            }
        };
    }

    // A random bean
    @Bean
    public LoanRequestBean loanRequestBean() {
        return new LoanRequestBean();
    }
}
```

JPA with Hibernate 4.2.x

在 Activiti 引擎的 serviceTask 或 listener 中使用 Hibernate 4.2.x JPA 时，需要添加 Spring ORM 这个额外的依赖。Hibernate 4.1.x及以下版本是不需要的。应该添加如下依赖：

```
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-orm</artifactId>
  <version>${org.springframework.version}</version>
</dependency>
```

Business archives 业务文档

为了部署流程，它们不得不包装在一个业务文档中。一个业务文档是Activiti 引擎部署的单元。一个业务文档相当与一个压缩文件，它包含BPMN2.0 流程，任务表单，规则和其他任意类型的文件。 大体上，业务文档是包含命名资源的容器。

当一个业务文档被部署，它将会自动扫描以 .bpmn20.xml 或者.bpmn 作为扩展名的 BPMN 文件。每个那样的文件都将会被解析并且可能会包含多个流程定义。

注意

业务归档中的 *Java* 类将不能够添加到类路径下。为了能够让流程运行，必须把存在于业务归档程中的流程定义使用的所有自定义的类（例如：*Java*服务任务或者实现事件的监听器）放在 *activiti* 引擎的类路径下

Deploying programmatically 编程式部署

通过一个压缩文件部署业务归档，它看起来像这样：

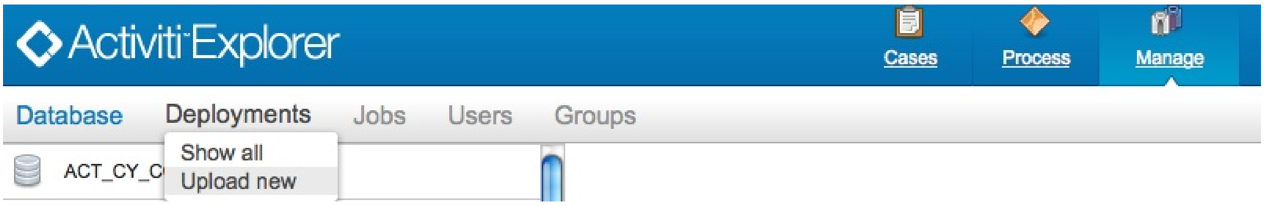
```
String barFileName = "path/to/process-one.bar";
ZipInputStream inputStream = new ZipInputStream(new FileInputStream(barFileName));

repositoryService.createDeployment()
    .name("process-one.bar")
    .addZipInputStream(inputStream)
    .deploy();
```

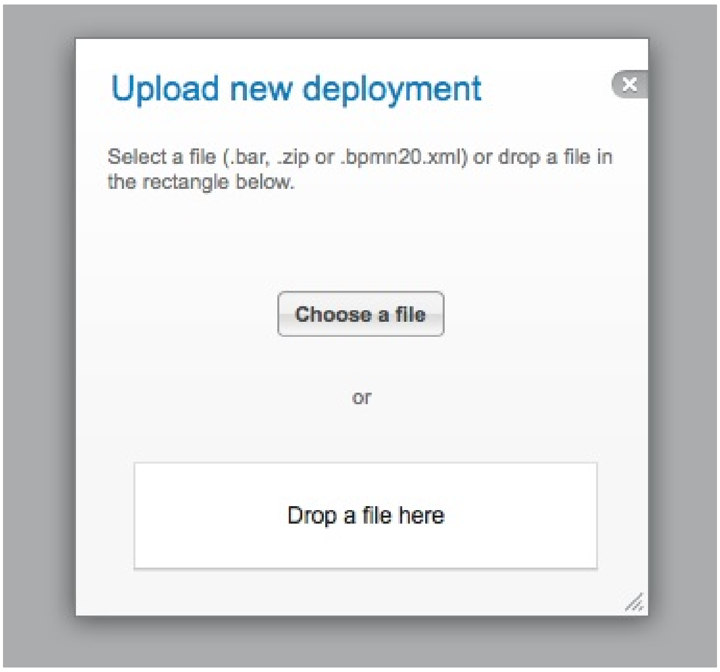
它也可以通过一个独立资源构建部署。 详细信息请查看javadocs。

Deploying with Activiti Explorer 通过 Activiti Explorer 控制台部署

Activiti web 控制台允许你通过 web 界面的用户接口上传一个 bar 格式的压缩文件（或者一个 bpmn20.xml 格式的文件）。选择 Management 标签,点击 Deployment:



现在将会有一个弹出窗口允许你从电脑上面选择一个文件，或者你可以简单的拖拽到指定的区域（如果你的浏览器支持）。


Upload new deployment
Select a file (.bar, .zip or .bpmn20.xml) or drop a file in the rectangle below.
Choose a file
or
Drop a file here

External resources 外部资源

流程定义保存在 Activiti 所支持的数据库中。当使用服务任务、执行监听器或者从 Activiti 配置文件中配置的 Spring beans 时，流程定义能够引用这些委托类。 这些类或者 Spring 配置文件对于所有流程引擎中可能执行的流程定义必须是可用的。

Java classes

当流程实例被启动的时候，在流程中被使用的所有自定义类（例如：服务任务中使用的 JavaDelegates、事件监听器、任务监听器,...）应该存在与流程引擎的类路径下。

然后，在部署业务文档时，这些类不必都存在于类路径下。当使用 Ant 部署一个新的业务文档时，这意味着你的委托类不必存在与类路径下。

当你使用示例设置并添加你自定义的类，你应该添加包含自定义类的 jar 包到 activitiexplorer 控制台或者 activiti-rest 的 webapp lib文件夹中。以及不要忽略包含你自定义类的依赖关系（如果有）。另外，你还可以包含你自己的依赖添加到你的 Tomcat容器的安装目录中的 ${tomcat.home}/lib。

Using Spring beans from a process 在流程中使用 Spring beans

当表达式或者脚本使用 Spring beans 时， 这些 beans 对于引擎执行流程定义时必须是可用的。如果你将要构建你自己的 web 应用并且按照Spring 集成这一章中描述那样在你的应用上下文配置流程引擎，这个看上去非常的简单。但是要记住，如果你也在使用 Activiti rest web 应用，那么也应该更新 Activiti rest web应用的上下文。 你可以把在 activiti-rest/lib/activiti-cfg.jar 文件中的 activiti.cfg.xml 替换成你的 Spring 上下文配置的 activiti-context.xml文件。

Creating a single app 创建独立应用

你可以考虑把 Activiti rest web 应用加入到你的 web 应用之中，因此，就仅仅只需要配置一个 ProcessEngine，从而不用确保所有的流程引擎的所有委托类在类路径下面并且是否使用正确的 spring 配置。

Versioning of process definitions 流程定义的版本

BPMN 中并没有版本的概念，没有版本也是不错的，因为可执行的 BPMN 流程作为你开发项目的一部分存在版本控制系统的知识库中（例如 SVN,Git 或者 Mercurial ）。 而在 Activiti 中，流程定义的版本是在部署时创建的。在部署的时候，流程定义被存储到 Activiti 使用的数据 库之前，Activiti 将会自动给 流程定义 分配一个版本号。

对于业务文档中每一个的流程定义，都会通过下列部署执行初始化属性 key, version, name 和 id:

- XML 文件中流程定义（流程模型）的 id 属性被当做是流程定义的 key属性。
- XML文件中的流程模型的 name 属性被当做是流程定义的 name 属性。如果该 name 属性并没有指定，那么 id 属性被当做是 name。
- 带有特定 key 的流程定义在第一次部署的时候，将会自动分配版本号为1，对于之后部署相同 key 的流程定义时候，这次部署的版本号将会设置为比当前最大的版本号大1的值。该 key 属性被用来区别不同的流程定义。
- 流程定义中的id属性被设置为 {processDefinitionKey}:{processDefinitionVersion}: {generated-id}, 这里的 generated-id 是一个唯一的数字被添加，用于确保在集群环境中缓存的流程定义的唯一性。

看下面示例

```
<definitions id="myDefinitions" >
  <process id="myProcess" name="My important process" >
    ...
```

当部署了这个流程定义之后，在数据库中的流程定义看起来像这样：

Table 6.1.

id	key	name	version
myProcess:1:676	myProcess	My important process	1

假设我们现在部署用一个流程的最新版本号（例如 改变用户任务），但是流程定义的 id 保持不变。 流程定义表将包含以下列表信息：

Table 6.2.

id	key	name	version
myProcess:1:676	myProcess	My important process	1
myProcess:2:870	myProcess	My important process	2

当 runtimeService.startProcessInstanceByKey("myProcess") 方法被调用时，它将会使用流程定义版本号为2的，因为这是最新版本的流程定义。可以说每次流程定义创建流程实例时，都会默认使用最新版本的流程定义。

我们应该创建第二个流程，在 Activiti 中，如下,定义并且部署它，该流程定义会添加到流程定义表中

```
<definitions id="myNewDefinitions" >
  <process id="myNewProcess" name="My important process" >
    ...
```

表格如下：

Table 6.3.

id	key	name	version
myProcess:1:676	myProcess	My important process	1
myProcess:2:870	myProcess	My important process	2
myNewProcess:1:1033	myNewProcess	My important process	1

注意：为何新流程的 key 与我们的第一个流程是不同的。尽管流程定义的名称是相同的（当然，我们应该也是可以改变这一点的）， Activiti 仅仅只考虑 id 属性判断流程。因此，新的流程定义部署的版本号为1。

Providing a process diagram 提供流程图

流程定义的流程图可以被添加到部署中，该流程图将会持久化到 Activiti所使用的数据库中并且可以通过 Activiti 的 API 进行访问。该流程图也可以被用来在 Activiti Explorer 控制台中的流程中进行显示。 如果在我们的类路径下面有一个流程，org/activiti/expenseProcess.bpmn20.xml， 该流程定义有 一个流程 key 'expense'。 以下遵循流程定义图片的命名规范（按照这个特地顺序）：

- 如果在部署时一个图片资源已经存在，它是 BPMN2.0 的 XML 文件名后面是流程定义的 key 并且是一个图片的后缀。那么该图片将被使用。在我们的例子中， 这应该是 org/activiti/expenseProcess.expense.png（或者 jpg/gif）。如果你在一个 BPMN2.0 XML 文件中定义多个流 程定义图片，这种方式更有意义。每个流程定义图片的文件名中都将会有一个流程定义 key。
- 如果并没有这样的图片存在，部署的时候寻找与匹配BPMN2.0 XML 文件的名称的图片资 源。在我们的例子中，这应该是 org/activiti/expenseProcess.png. 注意：这意味着在同一个 BPMN2.0 XML 文件夹中的每个流程定义都会有相同的流程定义图片。因此，在每一个 BPMN 2.0 XML 文件夹中仅仅只有一个流程定义，这绝对是不会有问题的。

当使用编程式的部署方式：

```
repositoryService.createDeployment()
  .name("expense-process.bar")
  .addClasspathResource("org/activiti/expenseProcess.bpmn20.xml")
  .addClasspathResource("org/activiti/expenseProcess.png")
  .deploy();
```

接下来，可以通过 API 来获取流程定义图片资源：

```
ProcessDefinition processDefinition = repositoryService.createProcessDefinitionQuery()
                                                         .processDefinitionKey("expense")
                                                         .singleResult();

  String diagramResourceName = processDefinition.getDiagramResourceName();
  InputStream imageStream = repositoryService.getResourceAsStream(processDefinition.getDeploymentId(), diagramResourceN
```

Generating a process diagram 生成流程图

在部署时没有提供图片的情况下，在 上一节中描述,如果流程定义中包含必要的'图像交换'信息时，Activiti 流程引擎竟会自动生成一个图像。

该资源可以按照部署时 提供流程图片完全相同的方式获取。



如果，因为某种原因，在部署的时候，并不需要或者不必要生成流程定义图片，那么就需要在流程引擎配置的属性中使用 isCreateDiagramOnDeploy：

```
<property name="createDiagramOnDeploy" value="false" />
```

现在图就不会生成了

Category 类别

部署和流程定义都是用户定义的类别。流程定义类别在 BPMN 文件中属性的初始化的值 <definitions ... targetNamespace="yourCategory" ... 部署类别是可以直接使用 API 进行指定的看起来想这样：

```
repositoryService
    .createDeployment()
    .category("yourCategory")
    ...
    .deploy();
```

What is BPMN 什么是 BPMN

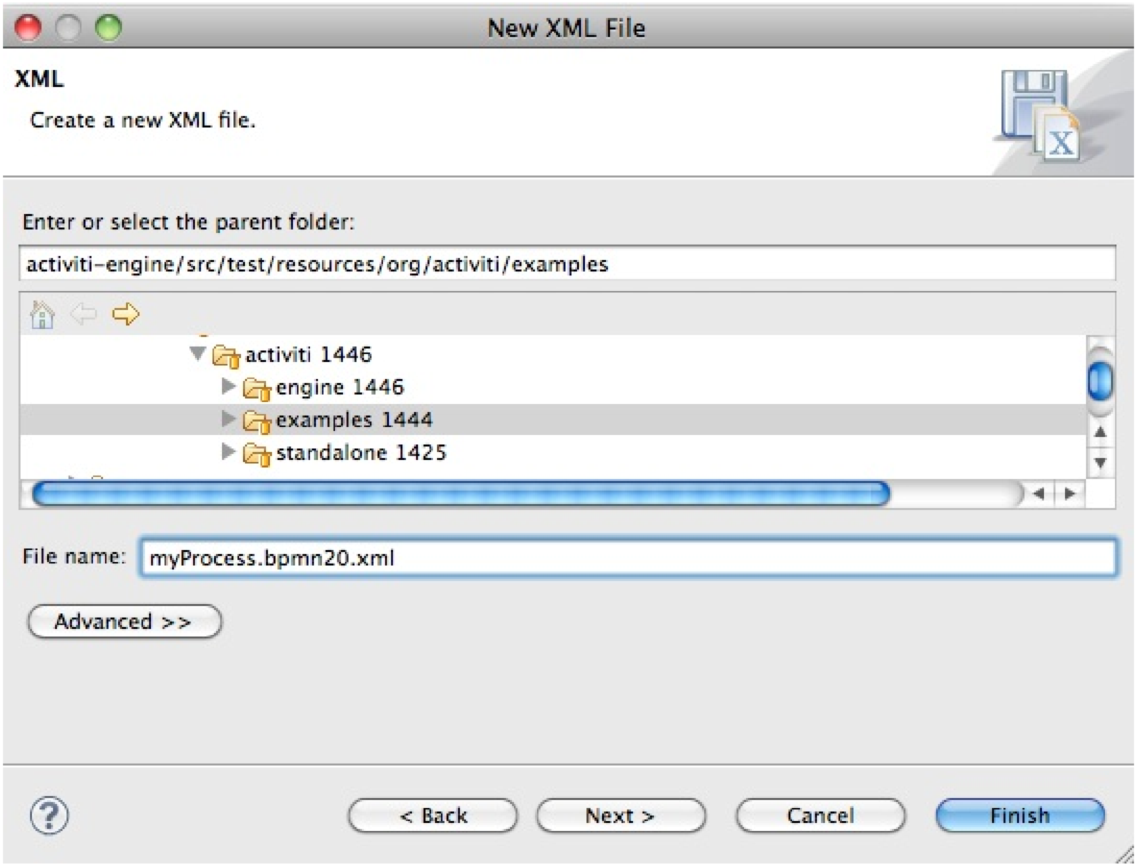
见FAQ 中关于 BPMN 2.0 部分

Defining a process 定义流程

注意

文章假设你在使用 *Eclipse IDE [http://eclipse.org/]*来创建和编辑文件。 不过，其中只用到了 *Eclipse* 很少的特性。你可以使用喜欢的任何工具来创建包含 *BPMN 2.0* 的 *xml* 文件。

创建一个新的 XML 文件（右击任何项目选择“新建”->“其他”->“XML-XML文件”）并命名。 确认文件后缀为 **.bpmn20.xml** 或 **.bpmn**， 否则引擎无法发布。



BPMN 2.0 根节点是 definitions 节点。 这个元素中，可以定义多个流程定义（不过我们建议每个文件只包含一个流程定义，可以简化开发过程中的维护难度）。 一个空的流程定义看起来像下面这样。注意，definitions 元素 最少也要包含 xmlns 和 targetNamespace 的声明。targetNamespace 可以是任意值，它用来对流程实例进行分类。

```
<definitions
  xmlns="http://www.omg.org/spec/BPMN/20100524/MODEL"
  xmlns:activiti="http://activiti.org/bpmn"
  targetNamespace="Examples">

  <process id="myProcess" name="My First Process">
    ..
  </process>

</definitions>
```

你也可以选择添加线上的 BPMN 2.0 格式位置， 下面是 ecilpse 中的xml 配置。

```
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.omg.org/spec/BPMN/20100524/MODEL
```

```
                    http://www.omg.org/spec/BPMN/2.0/20100501/BPMN20.xsd
```

process 元素有两个属性：

- id：这个属性是必须的， 它对应着 Activiti ProcessDefinition 对象的 key 属性。id 可以用来启动流程定义的流程实例，通过RuntimeService的startProcessInstanceByKey 方法。这个方法会一直使用最新发布版本的流程定义(译者注：实际中一般都使用这种方式启动流程)。

  ProcessInstance processInstance = runtimeService.startProcessInstanceByKey("myProcess");

注意，它和 startProcessInstanceById 方法不同。 这个方法期望使用Activiti 引擎在发布时自动生成的 id。可以通过调用processDefinition.getId() 方法获得这个值。 生成的 id 的格式为'key:version'， 最大长度限制为64个字符， 如果你在启动时抛出了一个 ActivitiException，说明生成的 id 太长了， 需要限制流程的key的长度。

- name：这个属性是可选的， 对应 ProcessDefinition 的 name 属性。 引擎自己不会使用这个属性，它可以用来在用户接口显示便于阅读的名称。

Getting started 10 minute tutorial 快速开始-十分钟教程

这张我们会演示一个（非常简单）的业务流程，我们会通过它介绍一些基本的 Activiti 概念和 API。

Prerequisites 前提

教程假设你已经能安装并运行 Activiti demo， 并且你使用了独立运行的 H2 服务器。修改 db.properties，设置其中的 jdbc.url=jdbc:h2:tcp://localhost/activiti，然后根据 H2 的文档启动独立服务器。

Goal 目标

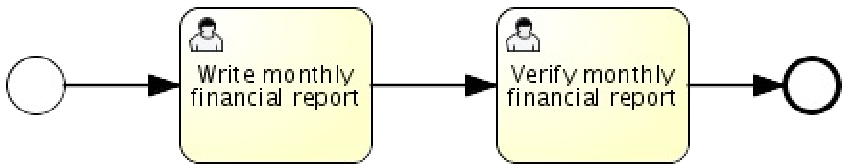
教程的目标是学习 Activiti 和一些基本的 BPMN 2.0 概念。 最终结果是一个简单的 Java SE 程序可以发布流程定义， 通过 Activiti 引擎API 操作流程。 我们也会使用一些 Activiti 相关的工具。当然，我们在教程中所学的 也可以用于你构建自己的业务流程 web 应用。

Use case 用例

用例很直接：我们有一个公司，就叫 BPMCorp。 在 BPMCorp 中，每个月都要给公司领导一个金融报表。 由会计部门负责。 当报表完成时，一个上级领导需要审批文档， 然后才能发给所有领导。

Process diagram 流程图

上面描述的业务流程可以用 Activiti Designer 进行可视化设计。 然后，为了这个教程，我们会手工编 写XML，这样可以学到更多知识细节。 我们流程的图形化 BPMN 2.0 标记看起来像这样：



我们看到有空开始事件（左侧圆圈），后面是两个用户任务： “制作月度财报”和 “验证月度财报”，最后是 空结束事件（右侧粗线圆圈）

XML representation 内容

业务流程的XML内容（FinancialReportProcess.bpmn20.xml）如下所示： 很容易找到流程的主要元素（点击链接可以了解 BPMN 2.0结构的详细信息）：

- （空）开始事件 是我们流程的入口。
- 个用户任务是流程中与操作者相关的任务声明。 注意第一个任务分配给 accountancy 组， 第二个任务分配给 management 组。 参考用户任务分配章节 了解更多关于用户任务分配人员和群组的问题。
- 当流程达到空结束事件就会结束。
- 这些元素都使用连线连接。 这些连线拥有 source 和 target 属性， 定义了连线的方向

  Write monthly financial report for publication to shareholders. accountancy Verify monthly financial report composed by the accountancy department. This financial report is going to be sent to all the company shareholders. management

Starting a process instance 启动一个流程实例

现在我们创建好了业务流程的流程定义。 有了这个流程定义，我们可以创建流程实例了。这时，一个流程实例对应了特定月度财报的创建和审批。 所有流程实例都共享同一个流程定义。

为了使用流程定义创建流程实例， 首先要发布业务流程， 这意味着两方面：

- 流程定义会保存到持久化的数据存储里， 是为你的 Activiti 引擎特别配置。所以部署好你的业务流程， 我们就能确认引擎重启后还能找到流程定义。
- BPMN 2.0 流程文件会解析成内存对象模型， 可以通过 Activiti API操作。可以通过发布章节获得关于发布的更多信息。就像章节里描述的一样，有很多种方式可以进行发布。 一种方式是通过下面的 API。注意所有与 Activiti 引擎的交互都是通过 services。

  Deployment deployment = repositoryService.createDeployment()

  ```
  .addClasspathResource("FinancialReportProcess.bpmn20.xml")
  .deploy();
  ```

现在我们可以启动一个新流程实例， 使用我们定义在流程定义里的 id（对应 XML 文件中的 process 元素）。 注意这里的 id 对于 Activiti 来说， 应该叫做 key（译者注：一般在流程模型中使用的 ID，在 Activiti 中都是 Key，比如任务 ID等...）。

```
ProcessInstance processInstance = runtimeService.startProcessInstanceByKey("financialReport");
```

这会创建一个流程实例，首先进入开始事件。 开始事件之后，它会沿着所有的外出连线（这里只有一条）执行， 到达第一个任务（“制作月度财报”）。 Activiti 会把一个任务保存到数据库里。 这时，分配到这个任务的用户或群组会被解析，也会保存到数据库里。 需要注意，Activiti 引擎会继续执行流程的环节，除非遇到一个 等待状态，比如用户任务。 在等待状态下，当前的流程实例的状态会保存到数据库中。 直到用户决定完成任务才能改变这个状态。这时，引擎会继续执行， 直到遇到下一个等待状态，或流程结束。 如果中间引擎重启或崩溃， 流程状态也会安全的保存在数据库里。

任务创建之后，startProcessInstanceByKey 会在到达用户任务 这个等待状态之后才会返回。这时，任务分配给了一个组，这意味着这个组是执行这个任务的候选组。

我们现在把所有东西都放在一起，来创建一个简单的java程序。 创建一个eclipse 项目，把 Activiti 的 jar 和依赖放到 classpath 下。 （这些都可以在 Activiti 发布包的 libs 目录下找到）。 在调用 Activiti 服务之前，我们必须构造一个 ProcessEngine， 它可以让我们访问服务。这里我们使用'单独运行'的配置， 这会使用 demo 安装时的数据库来构建 ProcessEngine。

你可以在这里下载流程定义 XML。 这个文件包含了上面介绍的 XML，也包含了必须的 BPMN 图像交换信息 以便在 Activiti 工具中能编辑流程。

```
public static void main(String[] args) {

  // Create Activiti process engine
  ProcessEngine processEngine = ProcessEngineConfiguration
    .createStandaloneProcessEngineConfiguration()
    .buildProcessEngine();

  // Get Activiti services
  RepositoryService repositoryService = processEngine.getRepositoryService();
  RuntimeService runtimeService = processEngine.getRuntimeService();

  // Deploy the process definition
  repositoryService.createDeployment()
    .addClasspathResource("FinancialReportProcess.bpmn20.xml")
    .deploy();

  // Start a process instance
  runtimeService.startProcessInstanceByKey("financialReport");
}
```

Task lists 任务列表

我们现在可以通过 TaskService 来获得任务了，添加以下逻辑：

```
List<Task> tasks = taskService.createTaskQuery().taskCandidateUser("kermit").list();
```

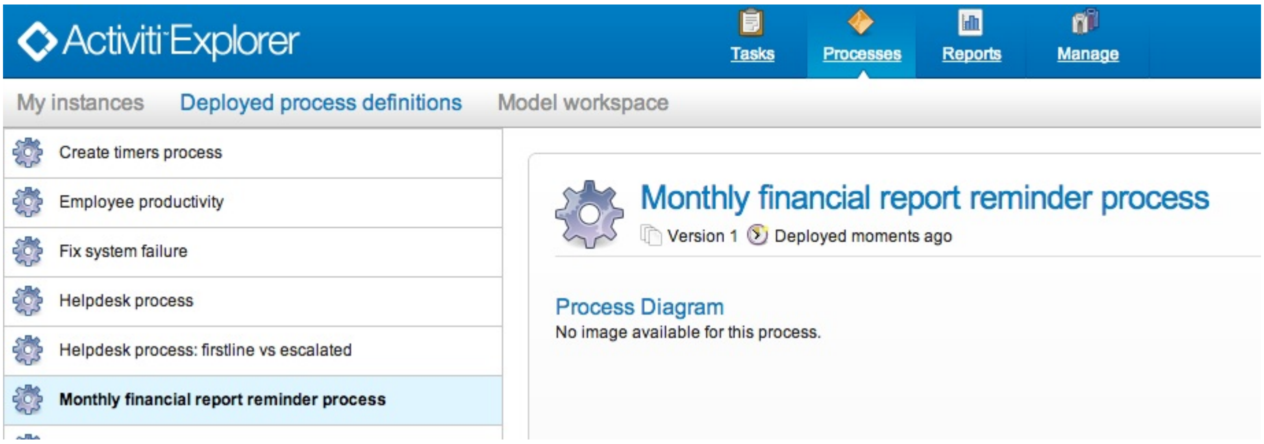
注意我们传入的用户必须是 accountancy 组的一个成员， 要和流程定义中向对应：

```
<potentialOwner>
    <resourceAssignmentExpression>
        <formalExpression>accountancy</formalExpression>
    </resourceAssignmentExpression>
</potentialOwner>
```

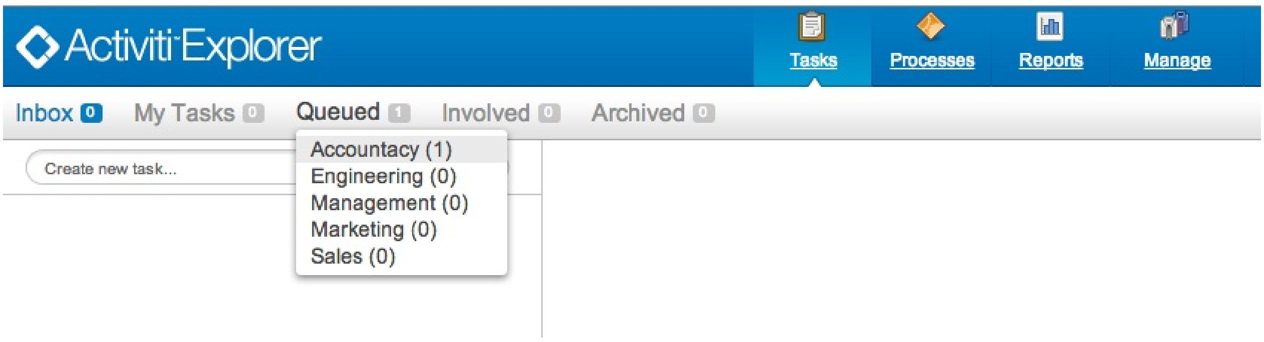
我们也可以使用群组名称，通过任务查询 API 来获得相关的结果。 现在可以在代码中添加如下逻辑：

```
TaskService taskService = processEngine.getTaskService();
List<Task> tasks = taskService.createTaskQuery().taskCandidateGroup("accountancy").list();
```

因为我们配置的 ProcessEngine 使用了与 demo 相同的数据， 我们可以登录到 Activiti Explorer。 默认，accountancy（会计）组里没有任何人。使用 kermit/kermit 登录，点击组，并创建一个新组。 然后点击用户，把组分配给 fozzie。现在使 用 fozzie/fozzie 登录，现在我们就可以启动我们的业务流程了， 选择 Processes 页，在'月度财报'的'操作'列 点击'启动流程'。



和上面介绍的那样，流程会执行到第一个用户任务。因为我们以 kermit 登录， 在启动流程实例之后，就可以看到有了一个新的待领任务。 选择任务页来查看这条新任务。 注意即使流程被其他人启动，任务还是会被会计组里的所有人作为一个候选任务看到。



Claiming the task 领取任务

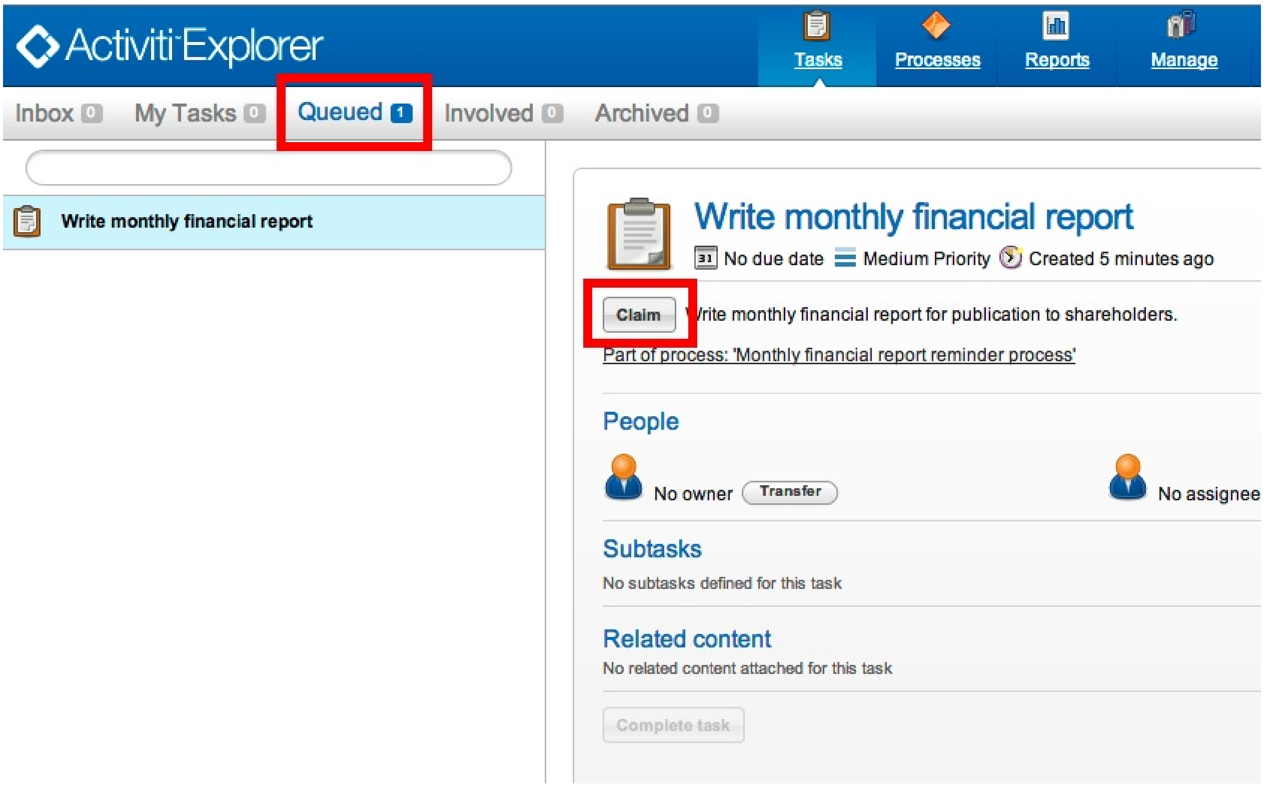
现在一个会计要认领这个任务。 认领以后，这个用户就会成为任务的执行人， 任务会从 accountancy 组的其他成员的任务列表中消失。 认领任务的代码如下所示：

```
taskService.claim(task.getId(), "fozzie");
```

任务会进入认领任务人的个人任务列表中。

```
List<Task> tasks = taskService.createTaskQuery().taskAssignee("fozzie").list();
```

在 Activiti Explorer UI 中，点击认领按钮，会执行相同的操作。 任务会移动到登录用户的个人任务列表。 你也会看到任务的执行人已经变成当前登陆的用户



Completing the task 完成任务

现在会计可以开始进行财报的工作了。报告完成后，他可以完成任务，意味着任务所需的所有工作都完成了

```
taskService.complete(task.getId());
```

对于 Activiti 引擎，需要一个外部信息来让流程实例继续执行。 任务会把自己从运行库中删除。 流程会沿着单独一个外出连线执行，移动到第二个任务（'审批报告'）。 与第一个任务相同的机制会使用到第二个任务上， 不同的是任务是分配给 management 组。

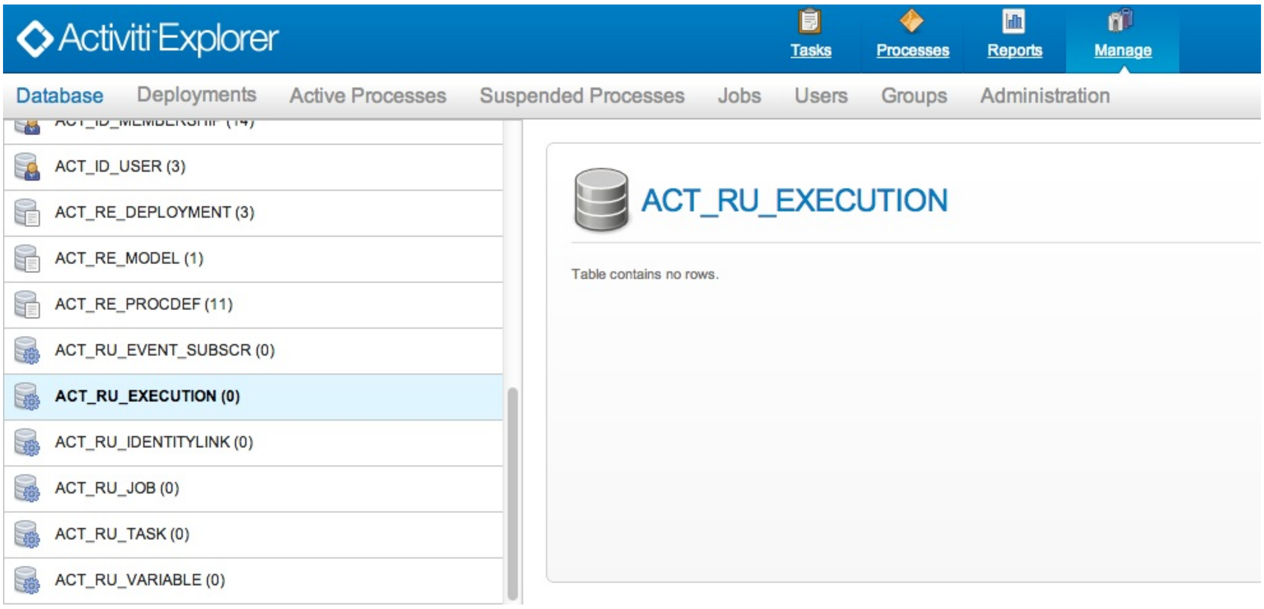
在 demo 中，完成任务是通过点击任务列表中的完成按钮。 因为 Fozzie 不是会计，我们先从 Activiti Explorer 注销 然后使用 kermit 登陆（他是经理）。 第二个任务会进入未分配任务列表。

Ending the process 结束流程

审批任务可以像之前介绍的一样查询和领取。 完成第二个任务会让流程执行到结束事件，就会结束流程实例。 流程实例和所

有相关的运行数据都会从数据库中删除。

登录 Activiti Explorer 就可以进行验证， 可以看到保存流程运行数据的表中已经没有数据了。



通过程序，你也可以使用 historyService 判断流程已经结束了。

```
HistoryService historyService = processEngine.getHistoryService();
HistoricProcessInstance historicProcessInstance =
historyService.createHistoricProcessInstanceQuery().processInstanceId(procId).singleResult();
System.out.println("Process instance end time: " + historicProcessInstance.getEndTime());
```

Code overview 代码总结

把上述代码组合在一起，获得的代码如下所示 （这些代码考虑到你可能会在 Activiti Explorer UI 中启动一些流程实例。 这样，它会获得多个任务，而不是一个， 所以代码可以一直正常运行） ：

```
public class TenMinuteTutorial {

  public static void main(String[] args) {

    // Create Activiti process engine
    ProcessEngine processEngine = ProcessEngineConfiguration
      .createStandaloneProcessEngineConfiguration()
      .buildProcessEngine();

    // Get Activiti services
    RepositoryService repositoryService = processEngine.getRepositoryService();
    RuntimeService runtimeService = processEngine.getRuntimeService();

    // Deploy the process definition
    repositoryService.createDeployment()
      .addClasspathResource("FinancialReportProcess.bpmn20.xml")
      .deploy();

    // Start a process instance
    String procId = runtimeService.startProcessInstanceByKey("financialReport").getId();

    // Get the first task
    TaskService taskService = processEngine.getTaskService();
    List<Task> tasks = taskService.createTaskQuery().taskCandidateGroup("accountancy").list();
    for (Task task : tasks) {
      System.out.println("Following task is available for accountancy group: " + task.getName());

      // claim it
```

```
        taskService.claim(task.getId(), "fozzie");
      }

      // Verify Fozzie can now retrieve the task
      tasks = taskService.createTaskQuery().taskAssignee("fozzie").list();
      for (Task task : tasks) {
        System.out.println("Task for fozzie: " + task.getName());

        // Complete the task
        taskService.complete(task.getId());
      }

      System.out.println("Number of tasks for fozzie: "
              + taskService.createTaskQuery().taskAssignee("fozzie").count());

      // Retrieve and claim the second task
      tasks = taskService.createTaskQuery().taskCandidateGroup("management").list();
      for (Task task : tasks) {
        System.out.println("Following task is available for accountancy group: " + task.getName());
        taskService.claim(task.getId(), "kermit");
      }

      // Completing the second task ends the process
      for (Task task : tasks) {
        taskService.complete(task.getId());
      }

      // verify that the process is actually finished
      HistoryService historyService = processEngine.getHistoryService();
      HistoricProcessInstance historicProcessInstance =
        historyService.createHistoricProcessInstanceQuery().processInstanceId(procId).singleResult();
      System.out.println("Process instance end time: " + historicProcessInstance.getEndTime());
    }

  }
```

这段代码包含在实例中的一个单元测试中（是的，你可以运行单元测试来测试你的流程。 参考单元测试章节来了解更多信息）

Future enhancements 未来的增强

可以看到业务流程相对于现实来说太简单了。 然而，你可以了解 Activiti中的 BPMN 2.0 结构， 你可以考虑对业务流程进行以下方面的加强：

- 定义网关来实现决策环节。 这样，经理可以驳回财报， 重新给会计创建一个任务。
- 考虑使用变量， 这样我们可以保存或引用报告， 把它显示到表单中。
- 在流程最后加入服务任务， 把报告发给每个领导。
- 其它

Chapter 8. BPMN 2.0 Constructs 关于 BPMN 2.0 架构

本章介绍 Activiti 支持的 BPMN 2.0 结构， 以及对 BPMN 标准的扩展。

Custom extensions 自定义扩展

BPMN 2.0 标准对于各方都是一个好东西。最终用户不用担心会绑死在供应商提供的专有解决方案上。 框架，特别是 Activiti 这样的开源框架，可以提供相同功能 （甚至是更好的实现），足以和大的供应商媲美。 按照BPMN 2.0 标准，从大供应商的解决方案迁移到 Activiti 只会经过一个简单而平滑的过程。

标准不好的一点是，它常常是不同公司之间大量讨论和妥协的结果。 （而且通常是愿景）。作为开发者去阅读流程定义的 BPMN 2.0 XML 时，有时会感觉用这种结构和方法去做事太麻烦了。 因此 Activiti 把简化开发作为最优先的事情，我们会使用一些被称为 'Activiti BPMN 扩展'的功能。这些扩展是新的结构或方法来简化对应的结构，它们并不属于 BPMN 2.0 规范。

虽然 BPMN 2.0 规范清楚的指明了如何开发自定义扩展， 但是我们还要确认一下几点：

- 自定义扩展的前提是总有简单的方法 转换成标准方法。 所以当你决定使用自定义扩展时，不用担心没办法回头。
- 当使用自定义扩展时，总会清楚的指明使用了 新的XML元素，属性，等等。 比如会使用 **activiti:** 命名空间前缀。
- 这些扩展的目标是最终把它们加入到下一版本的 BPMN 规范中，或者至少可以引起对特定 BPMN 结构的讨论。

因此无论是是否想要使用自定义扩展，这都取决于你。很多因素会影响决定这个决定（图形编辑器，公司策略，等等）。只是因为我们相信标准里的一些功能可以更简单或更高效，所以才决定提供自定义扩展。请对扩展给予我们（正面或负面）的评价， 或者是对自定义扩展的心想法。 说不定有一天你的想法就会加入到规范中

Events 事件

事件用来表明流程的生命周期中发生了什么事。 事件总是画成一个圆圈。 在BPMN 2.0 中，事件有两大分类：捕获（catching） 或 触发（throwing） 事件。

- 捕获（Catching）：当流程执行到事件， 它会等待被触发。触发的类型是由内部图表或 XML 中的类型声明来决定的。捕获事件与触发事件在显示方面是根据内部图表是否被填充来区分的（白色的）。
- 触发（Throwing）：当流程执行到事件， 会触发一个事件。触发的类型是由内部图表或 XML 中的类型声明来决定的。触发事件与捕获事件在显示方面是根据内部图表是否被填充来区分的（被填充为黑色）。

Event Definitions 事件定义

事件定义决定了事件的语义。如果没有事件定义，这个事件就不做什么特别的事情。 没有设置事件定义的开始事件不会在启动流程时做任何事情。如果给开始事件添加了一个事件定义（比如定时器事件定义）我们就声明了开始流程的事件 "类型 "（这时定时器事件监听器会在某个时间被触发）。

Timer Event Definitions 定时器事件定义

定时器事件是根据指定的时间触发的事件。可以用于 开始事件， 中间事件 或 边界事件。

定时器定义必须包含下面介绍的一个元素：

- timeDate。使用 ISO 8601 格式指定一个确定的时间，触发事件的时间。示例：

  2011-03-11T12:13:14
- timeDuration。指定定时器之前要等待多长时间， timeDuration 可以设置为 timerEventDefinition 的子元素。 使用ISO 8601 规定的格式 （由 BPMN 2.0 规定）。示例（等待10天）。

  P10D
- timeCycle。指定重复执行的间隔，可以用来定期启动流程实例，或为超时时间发送多个提醒。 timeCycl e元素可以使用两种格式。第一种是 ISO 8601 标准的格式。示例（重复3次，每次间隔10小时）：

  R3/PT10H

另外，你可以使用 cron 表达式指定 timeCycle，下面的例子是从整点开始，每5分钟执行一次：

```
0 0/5 * * * ?
```

请参考 cron 表达式 教程来了解如何使用

注意： 第一个数字表示秒，而不是像通常 *Unix cron* 中那样表示分钟。 重复的时间周期能更好的处理相对时间，它可以计算一些特定的时间点（比如，用户任务的开始时间），而 *cron* 表达式可以处理绝对时间 - 这对定时开始事件特别有用。

你可以在定时器事件定义中使用表达式，这样你就可以通过流程变量来影响那个定时器定义。 流程定义必须包含ISO 8601（或cron）格式的字符串，以匹配对应的时间类型。

```
<boundaryEvent id="escalationTimer" cancelActivity="true" attachedToRef="firstLineSupport">
     <timerEventDefinition>
      <timeDuration>${duration}</timeDuration>
    </timerEventDefinition>
  </boundaryEvent>
```

注意： 只有启用 job 执行器之后，定时器才会被触发。 （activiti.cfg.xml 中的 jobExecutorActivate 需要设置为 true， 不过，默认 job 执行器是关闭的）。

Error Event Definitions 错误事件定义

错误事件是由指定错误触发的。

重要提醒：BPMN 错误与 Java 异常完全不一样。 实际上，他俩一点儿共同点都没有。BPMN 错误事件是为了对业务异常建模。Java 异常是要用特定方式处理。

错误事件定义会引用一个 error 元素。下面是一个 error 元素的例子，引用了一个错误声明：

```
<endEvent id="myErrorEndEvent">
  <errorEventDefinition errorRef="myError" />
</endEvent>
```

引用相同 error 元素的错误事件处理器会捕获这个错误。

Signal Event Definitions 信号事件定义

信号事件会引用一个已命名的信号。信号全局范围的事件（广播语义）。 会发送给所有激活的处理器。

信号事件定义使用 signalEventDefinition 元素。 signalRef 属性会引用 definitions 根节点里定义的 signal 子元素。 下面是一个流程的实例，其中会抛出一个信号，并被中间事件捕获。

```
<definitions... >
        <!-- declaration of the signal -->
        <signal id="alertSignal" name="alert" />

        <process id="catchSignal">
                <intermediateThrowEvent id="throwSignalEvent" name="Alert">
                        <!-- signal event definition -->
                        <signalEventDefinition signalRef="alertSignal" />
                </intermediateThrowEvent>
                ...
                <intermediateCatchEvent id="catchSignalEvent" name="On Alert">
                        <!-- signal event definition -->
                        <signalEventDefinition signalRef="alertSignal" />
                </intermediateCatchEvent>
                ...
        </process>
</definitions>
```

signalEventDefinition 引用相同的 signal 元素。

Throwing a Signal Event 触发信号事件

既可以通过 bpmn 节点由流程实例触发一个信号，也可以通过 API 触发。 下面的 org.activiti.engine.RuntimeService 中的方法 可以用来手工触发一个信号。

```
RuntimeService.signalEventReceived(String signalName);
RuntimeService.signalEventReceived(String signalName, String executionId);
```

signalEventReceived(String signalName); 和 signalEventReceived(String signalName, String executionId); 之间的区别是 第一个方法会把信号发送给全局所有订阅的处理器（广播语义）， 第二个方法只把信息发送给指定的执行。

Catching a Signal Event 捕获信号事件

信号事件可以被中间捕获信号事件或边界信息事件捕获。

Querying for Signal Event subscriptions 查询信号事件的订阅

可以查询所有订阅了特定信号事件的执行：

```
List<Execution> executions = runtimeService.createExecutionQuery()
      .signalEventSubscriptionName("alert")
      .list();
```

我们可以使用 signalEventReceived(String signalName, String executionId) 方法把信号发送给这些执行。

Signal event scope 信号事件范围

默认，信号会在流程引擎范围内进行广播。就是说，你可以在一个流程实例中抛出一个信号事件，其他不同流程定义的流程实例都可以监听到这个事件。

然而，有时只希望在同一个流程实例中响应这个信号事件。 比如一个场景是，流程实例中的同步机制，如果两个或更多活动是互斥的。

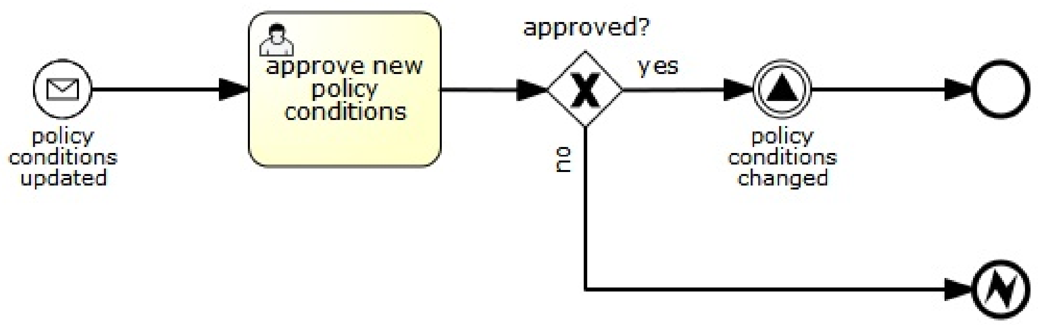
如果想要限制信号事件的范围，可以使用信号事件定义的 scope 属性（不是 BPMN2.0 的标准属性）：

```
<signal id="alertSignal" name="alert" activiti:scope="processInstance"/>
```

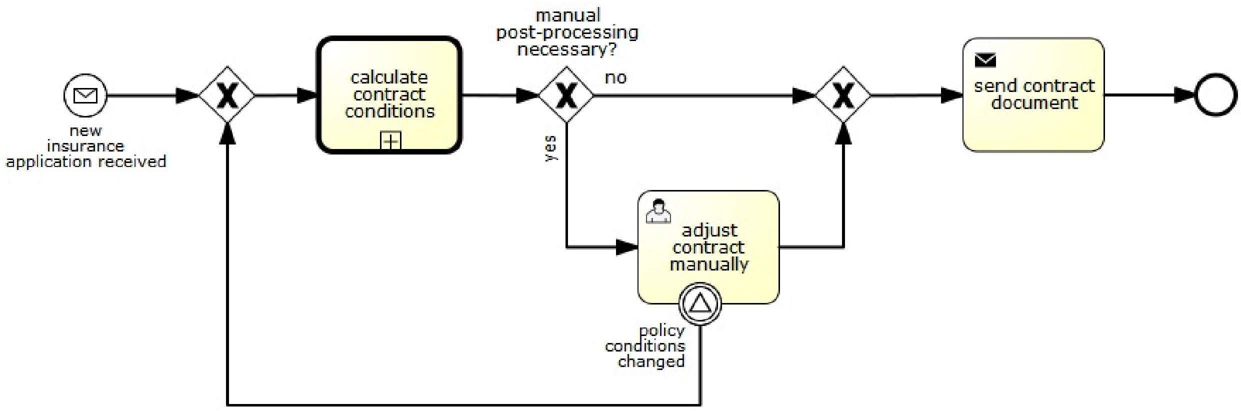
这个属性值默认是 "global"

Signal Event example(s) 信号事件实例

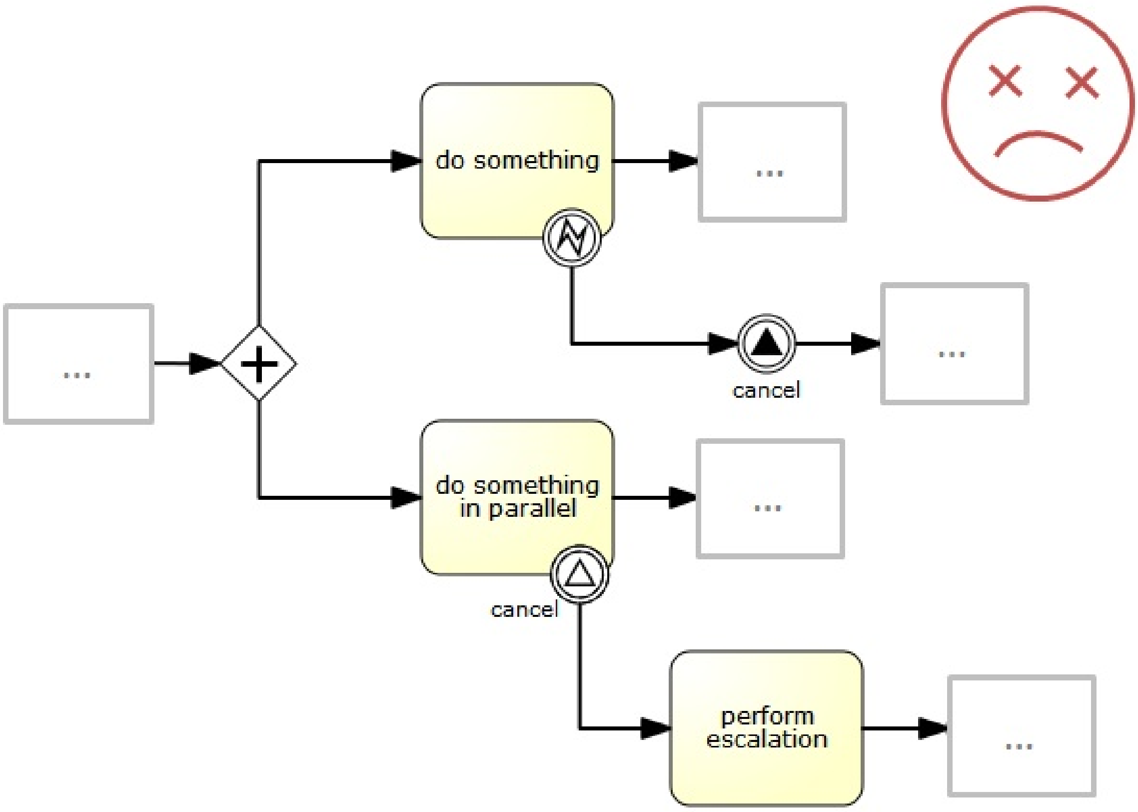
下面是两个不同流程使用信号交互的例子。第一个流程在保险规则更新或改变时启动。 在修改被参与者处理时，会触发一个信息，通知规则改变：



这个时间会被所有感兴趣的流程实例捕获。下面是一个订阅这个事件的流程实例。



注意：要了解信号事件是广播给所有激活的处理器的。 这意味着在上面的例子中，所有流程实例都会接收到这个事件。 这就是我们想要的。然而，有的情况下并不想要这种广播行为。 考虑下面的流程



上述流程描述的模式 activiti 并不支持。这种想法是执行“do something”任务时出现的错误，会被边界错误事件捕获，然后使用信号传播给并发路径上的分支，进而中断"do something inparallel"任务。 目前，activiti 实际运行的结果与期望一致。信号会传播给边界事件并中断任务。但是，根据信号的广播含义，它也会传播给所有其他订阅了信号事件的流程实例。 所以，这就不是我们想要的结果。

注意： 信号事件不会执行任何与特定流程实例的联系。 如果你只想把一个信息发给指定的流程实例，需要手工关联，再使用 signalEventReceived(String signalName, String executionId) 和对应的查询机制。

Message Event Definitions 消息事件定义

消息事件的事件消息引用命名。一个消息有一个名字和一个有效载荷。不像一个信号，一个消息事件总是指向一个接收器。

一个消息事件的定义是使用 messageeventdefinition 元素声明。属性messageref 引用一个消息元素声明为定义的根元素的子元素。以下是摘录的一个过程，其中两个消息事件声明的开始事件和中间捕捉事件消息引用。

```
<definitions id="definitions"
  xmlns="http://www.omg.org/spec/BPMN/20100524/MODEL"
  xmlns:activiti="http://activiti.org/bpmn"
  targetNamespace="Examples"
  xmlns:tns="Examples">

  <message id="newInvoice" name="newInvoiceMessage" />
  <message id="payment" name="paymentMessage" />

  <process id="invoiceProcess">

    <startEvent id="messageStart" >
        <messageEventDefinition messageRef="newInvoice" />
    </startEvent>
    ...
    <intermediateCatchEvent id="paymentEvt" >
        <messageEventDefinition messageRef="payment" />
    </intermediateCatchEvent>
    ...
  </process>

</definitions>
```

Throwing a Message Event 触发消息事件

作为一个嵌入式的流程引擎，activit i不能真正接收一个消息。这些环境相关，与平台相关的活动 比如连接到 JMS（Java消息服务）队列或主题或执行 WebService 或 REST 请求。 这个消息的接收是你要在应用或架构的一层实现的，流程引擎则内嵌其中。

在你的应用接收一个消息之后，你必须决定如何处理它。 如果消息应该触发启动一个新流程实例， 在下面的 RuntimeService 的两个方法中选择一个执行：

```
ProcessInstance startProcessInstanceByMessage(String messageName);
ProcessInstance startProcessInstanceByMessage(String messageName, Map<String, Object> processVariables);
ProcessInstance startProcessInstanceByMessage(String messageName, String businessKey, Map<String, Object> processVariat
```

这些方法允许使用对应的消息系统流程实例。

如果消息需要被运行中的流程实例处理，首先要根据消息找到对应的流程实例 （参考下一节）然后触发这个等待中的流程。RuntimeService 提供了如下方法可以基于消息事件的订阅来触发流程继续执行

```
void messageEventReceived(String messageName, String executionId);
void messageEventReceived(String messageName, String executionId, HashMap<String, Object> processVariables);
```

Querying for Message Event subscriptions 查询消息事件的订阅

Activiti 支持消息开始事件和中间消息事件。

- 消息开始事件的情况，消息事件订阅分配给一个特定的 process definition 。这个消息订阅可以使用 ProcessDefinitionQuery 查询到：

  ProcessDefinition processDefinition = repositoryService.createProcessDefinitionQuery()

  ```
      .messageEventSubscription("newCallCenterBooking")
      .singleResult();
  ```

因为同时只能有一个流程定义关联到消息的订阅点，查询总是返回0或一个结果。 如果流程定义更新了， 那么只有最新版本的流程定义会订阅到消息事件上。

- 中间捕获消息事件的情况，消息事件订阅会分配给特定的执行。 这个消息事件订阅可以使用 ExecutionQuery 查询到：

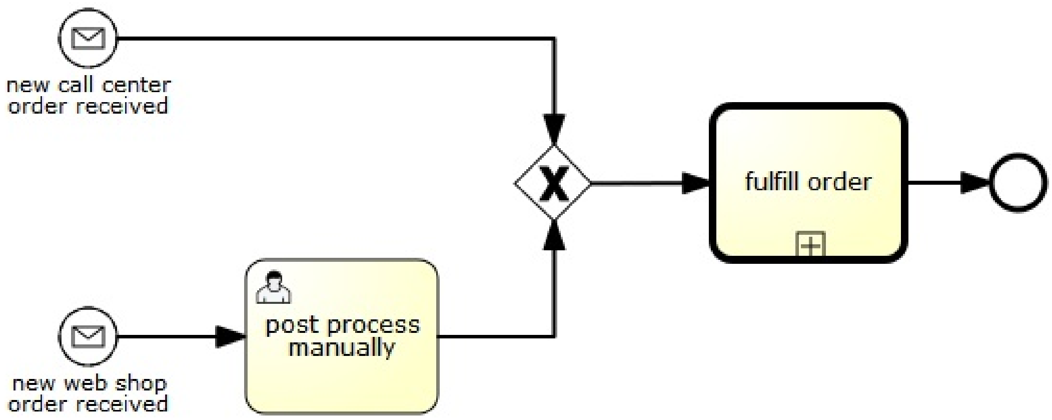
  Execution execution = runtimeService.createExecutionQuery()

```
      .messageEventSubscriptionName("paymentReceived")
      .variableValueEquals("orderId", message.getOrderId())
      .singleResult();
```

这个查询可以调用对应的查询，通常是流程相关的信息 （这里，最多只能有一个流程实例对应着 orderId ）。

Message Event example(s) 消息事件实例

下面是一个使用两个不同消息启动的流程实例：



可以用在，流程需要不同的方式来区分开始事件，而后最终会进入同样的路径。

Start Events 开始事件

开始事件用来指明流程在哪里开始。开始事件的类型（流程在接收事件时启动， 还是在指定时间启动，等等），定义了流程如何启动， 这通过事件中不同的小图表来展示。 在 XML 中，这些类型是通过声明不同的子元素来区分的。

开始事件都是捕获事件： 最终这些事件都是（一直）等待着，直到对应的触发时机出现。

在开始事件中，可以设置下面的 activiti 特定属性：

- initiator：当流程启动时，把当前登录的用户保存到哪个变量名中。 示例如下：

登录的用户必须使用 IdentityService.setAuthenticatedUserId(String) 方法设置， 并像这样包含在 try-finally 代码中：

```
try {
  identityService.setAuthenticatedUserId("bono");
  runtimeService.startProcessInstanceByKey("someProcessKey");
} finally {
  identityService.setAuthenticatedUserId(null);
}
```

这段代码来自 Activiti Explorer，所以它可以和Chapter 9. Forms 表单一起结合使用。

None Start Event 空开始事件

Description 描述

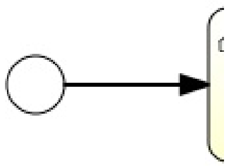
空开始事件技术上意味着没有指定启动流程实例的触发条件。 这就是说引擎不能预计什么时候流程实例会启动。 空开始事件用于，当流程实例要通过API 启动的场景， 通过调用 startProcessInstanceByXXX 方法。

```
ProcessInstance processInstance = runtimeService.startProcessInstanceByXXX();
```

注意： 子流程都有一个空开始事件。

Graphical notation 图形标记

空开始事件显示成一个圆圈，没有内部图表（没有触发类型）



XML representation 内容

空开始事件的XML结构是普通的开始事件定义，没有任何子元素 （其他开始事件类型都有一个子元素来声明自己的类型）

```
<startEvent id="start" name="my start event" />
```

Custom extensions for the none start event 空开始事件的自定义扩展

formKey：引用用户在启动新流程实例时需要填写的表单模板， 更多信息可以参考Chapter 9. Forms 表单。 实例：

```
<startEvent id="request" activiti:formKey="org/activiti/examples/taskforms/request.form" />
```

Timer Start Event 定时开始事件

Description 描述

定时开始事件用来在指定的时间创建流程实例。 它可以同时用于只启动一次的流程和应该在特定时间间隔启动多次的流程。

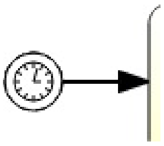
注意：子流程不能使用定时开始事件。

注意：定时开始事件在流程发布后就会开始计算时间。 不需要调用 startProcessInstanceByXXX， 虽然也而已调用启动流程的方法， 但是那会导致调用 startProcessInstanceByXXX 时启动过多的流程。

注意：当包含定时开始事件的新版本流程部署时， 对应的上一个定时器就会被删除。这是因为通常不希望自动启动旧版本流程的流程实例。

Graphical notation 图形标记

定时开始事件显示为了一个圆圈，内部是一个表。



XML representation 内容

定时开始事件的XML内容是普通开始事件的声明，包含一个定时定义子元素。 请参考Timer Event Definitions 定时器事件定义 查看配合细节。

示例：流程会启动4次，每次间隔5分钟，从2011年3月11日，12:13开始计时。

```
<startEvent id="theStart">
        <timerEventDefinition>
            <timeCycle>R4/2011-03-11T12:13/PT5M</timeCycle>
        </timerEventDefinition>
 </startEvent>
```

示例：流程会根据选中的时间启动一次。

```
<startEvent id="theStart">
        <timerEventDefinition>
            <timeDate>2011-03-11T12:13:14</timeDate>
        </timerEventDefinition>
</startEvent>
```

Message Start Event

Description 描述

消息开始事件可以用其使用一个命名的消息来启动流程实例。 这样可以帮助我们使用消息名称来选择正确的开始事件。

在发布包含一个或多个消息开始事件的流程定义时，需要考虑下面的条件：

- 消息开始事件的名称在给定流程定义中不能重复。流程定义不能包含多个名称相同的消息开始事件。 如果两个或以上消息开始事件应用了相同的事件，或两个或以上消息事件引用的消息名称相同，activiti 会在发布流程定义时抛出异常。
- 消息开始事件的名称在所有已发布的流程定义中不能重复。 如果一个或多个消息开始事件引用了相同名称的消息，而这个消息开始事件已经部署到不同的流程定义中， activiti 就会在发布时抛出一个异常。
- 流程版本：在发布新版本的流程定义时，之前订阅的消息订阅会被取消。 如果新版本中没有消息事件也会这样处理。

启动流程实例，消息开始事件可以使用 下列 RuntimeService 中的方法来触发：

```
ProcessInstance startProcessInstanceByMessage(String messageName);
ProcessInstance startProcessInstanceByMessage(String messageName, Map<String, Object> processVariables);
ProcessInstance startProcessInstanceByMessage(String messageName, String businessKey, Map<String, Object< processVariat
```

这里的 messageName 是 messageEventDefinition 的 messageRef 属性引用的 message 元素的 name 属性。 启动流程实例时，要考虑一下因素：

- 消息开始事件只支持顶级流程。消息开始事件不支持内嵌子流程。
- 如果流程定义有多个消息开始事件，runtimeService.startProcessInstanceByMessage(...) 会选择 对应的开始事件。

- 如果流程定义有多个消息开始事件和一个空开始事件。 runtimeService.startProcessInstanceByKey(...) 和 runtimeService.startProcessInstanceById(...) 会使用空开始事件启动流程实例。
- 如果流程定义有多个消息开始事件，而且没有空开始事件， runtimeService.startProcessInstanceByKey(...) 和 runtimeService.startProcessInstanceById(...) 会抛出异常。
- 如果流程定义只有一个消息开始事件， runtimeService.startProcessInstanceByKey(...) 和 runtimeService.startProcessInstanceById(...) 会使用这个消息开始事件启动流程实例。
- 如果流程被调用环节（callActivity）启动，消息开始事件只支持如下情况：
- 在消息开始事件以外，还有一个单独的空开始事件
- 流程只有一个消息开始事件，没有空开始事件。

Graphical notation 图形标记

消息开始事件是一个圆圈，中间是一个消息事件图标。图标是白色未填充的，来表示捕获（接收）行为。



XML representation 内容

消息开始事件的XML内容时在普通开始事件申请中包含一个 messageEventDefinition 子元素：

```
<definitions id="definitions"
  xmlns="http://www.omg.org/spec/BPMN/20100524/MODEL"
  xmlns:activiti="http://activiti.org/bpmn"
  targetNamespace="Examples"
  xmlns:tns="Examples">

  <message id="newInvoice" name="newInvoiceMessage" />

  <process id="invoiceProcess">

    <startEvent id="messageStart" >
        <messageEventDefinition messageRef="tns:newInvoice" />
    </startEvent>
    ...
  </process>

</definitions>
```

Signal Start Event

Description 描述

signal 开始事件，可以用来通过一个已命名的信号（signal）来启动一个流程实例。 信号可以在流程实例内部使用"中间信号抛出事务"触发， 也可以通过 API (runtimService.signalEventReceivedXXX 方法) 触发。两种情况下， 所有流程实例中拥有相同名称的 signalStartEvent 都会启动。

注意，在两种情况下，都可以选择同步或异步的方式启动流程实例。

必须向 API 传入 signalName， 这是 signal 元素的 name 属性值， 它会被 signalEventDefinition 的 signalRef 属性引用。

Graphical notation 图形标记

信号开始事件显示为一个中间包含信号事件图标的圆圈。标记是无填充的，表示捕获（接收）行为



XML representation 内容

signalStartEvent 的 XML 格式是标准的 startEvent 声明，其中包含一个 signalEventDefinition 子元素：

```
<process id="processWithSignalStart1">
    <startEvent id="theStart">
      <signalEventDefinition id="theSignalEventDefinition" signalRef="theSignal"  />
    </startEvent>
    <sequenceFlow id="flow1" sourceRef="theStart" targetRef="theTask" />
    <userTask id="theTask" name="Task in process A" />
    <sequenceFlow id="flow2" sourceRef="theTask" targetRef="theEnd" />
    <endEvent id="theEnd" />
</process>
```

Error Start Event

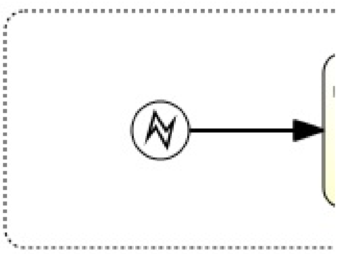
Description 描述

错误开始事件可以用来触发一个事件子流程。 错误开始事件不能用来启动流程实例。

错误开始事件都是中断事件。

Graphical notation 图形标记

错误开始事件是一个圆圈，包含一个错误事件标记。标记是白色未填充的，来表示捕获（接收）行为。



XML representation 内容

错误开始事件的 XML 内容是普通开始事件定义中，包含一个 errorEventDefinition 子元素。

```
<startEvent id="messageStart" >
        <errorEventDefinition errorRef="someError" />
</startEvent>
```

End Events 结束事件

结束事件表示（子）流程（分支）的结束。 结束事件都是触发事件。 这是说当流程达到结束事件，会触发一个结果。 结果的类型是通过事件的内部黑色图标表示的。 在XML内容中，是通过包含的子元素声明的。

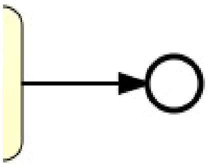
None End Event 空结束事件

Description 描述

空结束事件意味着到达事件时不会指定抛出的结果。 这样，引擎会直接结束当前执行的分支，不会做其他事情

Graphical notation 图形标记

空结束事件是一个粗边圆圈，内部没有小图表（无结果类型）



XML representation 内容

空结束事件的XML内容是普通结束事件定义，不包含子元素(其他结束事件类型都会包含声明类型的子元素)

```
<endEvent id="end" name="my end event" />
```

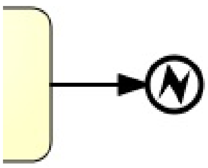
Error End Event 错误结束事件

Description 描述

当流程执行到错误结束事件，流程的当前分支就会结束，并抛出一个错误。 这个错误可以被对应的中间边界错误事件捕获。 如果找不到匹配的边界错误事件，就会抛出一个异常。

Graphical notation 图形标记

错误结束事件是一个标准的结束事件（粗边圆圈），内部有错误图标。 错误图表是全黑的，表示触发语法。



XML representation 内容

错误结束事件的内容是一个错误事件， 子元素为errorEventDefinition。

```
<endEvent id="myErrorEndEvent">
  <errorEventDefinition errorRef="myError" />
</endEvent>
```

errorRef 属性引用定义在流程外部的 error 元素：

```
<error id="myError" errorCode="123" />
...
<process id="myProcess">
...
```

error 的 errorCode 用来查找 匹配的捕获边界错误事件。 如果 errorRef 与任何 error 都不匹配， 就会使用errorRef来作为errorCode 的缩写。 这是 activiti 特定的缩写。 更具体的 说， 见如下代码：

```
<error id="myError" errorCode="error123" />
...
<process id="myProcess">
...
  <endEvent id="myErrorEndEvent">
    <errorEventDefinition errorRef="myError" />
  </endEvent>
```

等价于

```
<endEvent id="myErrorEndEvent">
  <errorEventDefinition errorRef="error123" />
</endEvent>
```

注意 errorRef 必须与 BPMN 2.0 格式相符， 必须是一个合法的 QName。

Cancel End Event 取消结束事件

[试验]

Description 描述

取消结束事件只能与BPMN事务子流程结合使用。 当到达取消结束事件时，会抛出取消事件，它必须被取消边界事件捕获。 取消边界事件会取消事务，并触发补偿机制。

Graphical notation 图形标记

取消结束事件显示为标准的结束事件（粗边圆圈），包含一个取消图标。 取消图标是全黑的，表示触发语法



XML representation 内容

取消结束事件内容是一个结束事件， 包含 cancelEventDefinition 子元素。

```
<endEvent id="myCancelEndEvent">
  <cancelEventDefinition />
</endEvent>
```

Boundary Events 边界事件

边界事件都是捕获事件，它会附在一个环节上。 （边界事件不可能触发事件）。这意味着，当节点运行时， 事件会监听对应

的触发类型。 当事件被捕获， 节点就会中断， 同时执行事件的后续连线。

所以边界事件的定义方式都一样：

```
<boundaryEvent id="myBoundaryEvent" attachedToRef="theActivity">
      <XXXEventDefinition/>
</boundaryEvent>
```

边界事件使用如下方式进行定义：

- 唯一标识（流程范围）
- 使用 caught 属性 引用事件衣服的节点。 注意边界事件和它们附加的节点在同一级别上。（比如，边界事件不是包含在节点内的）。
- 格式为 XXXEventDefinition 的 XML 子元素 （比如，TimerEventDefinition，ErrorEventDefinition，等等） 定义了边界事件的类型。参考对应的边界事件类型， 获得更多细节。

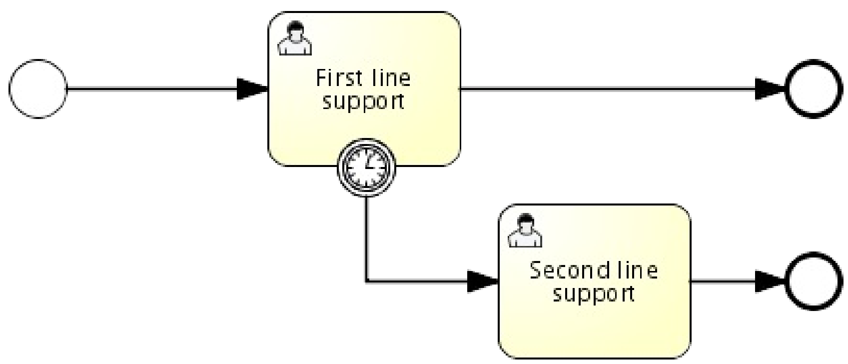
Timer Boundary Event 定时边界事件

Description 描述

定时边界事件就是一个暂停等待警告的时钟。当流程执行到绑定了边界事件的环节， 会启动一个定时器。 当定时器触发时（比如，一定时间之后）， 环节就会中断， 并沿着定时边界事件的外出连线继续执行。

Graphical notation 图形标记

定时边界事件是一个标准的边界事件（边界上的一个圆圈）， 内部是一个定时器小图标。



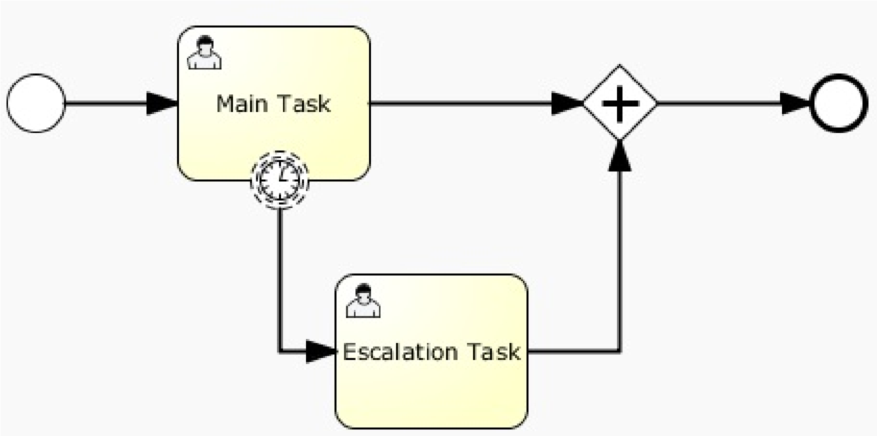
XML representation 内容

定时器边界任务定义是一个正规的边界事件。 指定类型的子元素是 timerEventDefinition 元素。

```
<boundaryEvent id="escalationTimer" cancelActivity="true" attachedToRef="firstLineSupport">
   <timerEventDefinition>
    <timeDuration>PT4H</timeDuration>
  </timerEventDefinition>
</boundaryEvent>
```

请参考定时事件定义获得更多定时其配置的细节。

在流程图中，可以看到上述例子中的圆圈边线是虚线：



经典场景是发送一个升级邮件，但是不打断正常流程的执行。

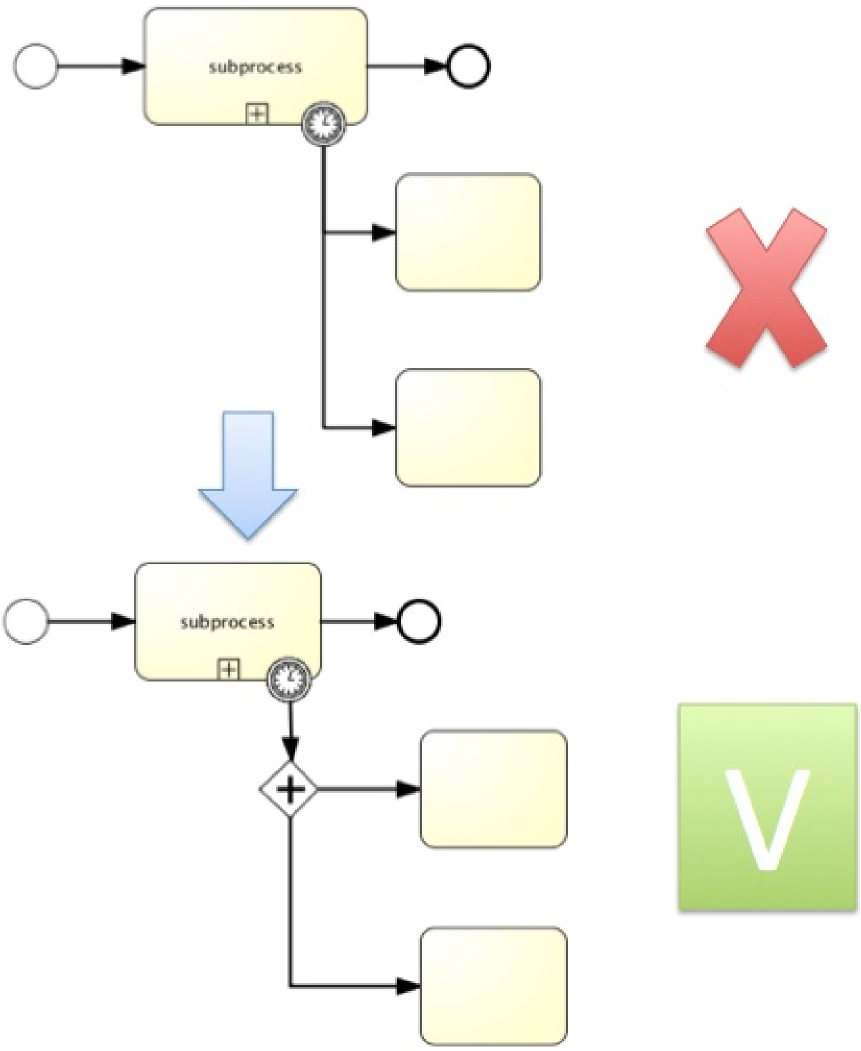
因为 BPMN 2.0 中，中断和非中断的事件还是有区别的。默认是中断事件。 非中断事件的情况， 不会中断原始环节，那个环节还停留在原地。 对应的，会创建一个新分支，并沿着事件的流向继续执行。 在XML内容中，要把cancelActivity 属性设置为 false：

```
<boundaryEvent id="escalationTimer" cancelActivity="false" attachedToRef="firstLineSupport"/>
```

注意：边界定时事件只能在job执行器启用时使用。（比如，把activiti.cfg.xml 中的 jobExecutorActivate 设置为 true，因为默认 job 执行器默认是禁用的）

Known issue with boundary events 边界事件的已知问题

使用边界事件有一个已知的同步问题。 目前，不能边界事件后面不能有多条外出连线 （参考ACT-47。 解决这个问题的方法是在一个连线后使用并发网关



Error Boundary Event 错误边界事件

Description 描述

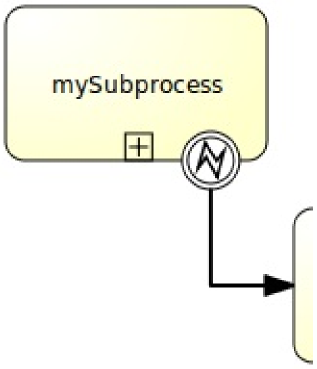
节点边界上的中间捕获错误事件， 或简写成边界错误事件，它会捕获节点范围内抛出的错误。

定义一个边界错误事件，大多用于内嵌子流程，或调用节点， 对于子流程的情况，它会为所有内部的节点创建一个作用范围。 错误是由错误结束事件抛出的。 这个错误会传递给上层作用域，直到找到一个错误事件定义向匹配的边界错误事件。

当捕获了错误事件时，边界任务绑定的节点就会销毁， 也会销毁内部所有的执行分支 （比如，同步节点，内嵌子流程，等等）。 流程执行会继续沿着边界事件的外出连线继续执行。

Graphical notation 图形标记

边界错误事件显示成一个普通的中间事件（圆圈内部有一个小圆圈） 放在节点的标记上，内部有一个错误小图标。错误小图标是白色的，表示它是一个捕获事件。



XML representation 内容

边界错误事件定义为普通的边界事件：

```
<boundaryEvent id="catchError" attachedToRef="mySubProcess">
  <errorEventDefinition errorRef="myError"/>
</boundaryEvent>
```

和错误结束事件一样， errorRef 引用了 process 元素外部的一个错误定义：

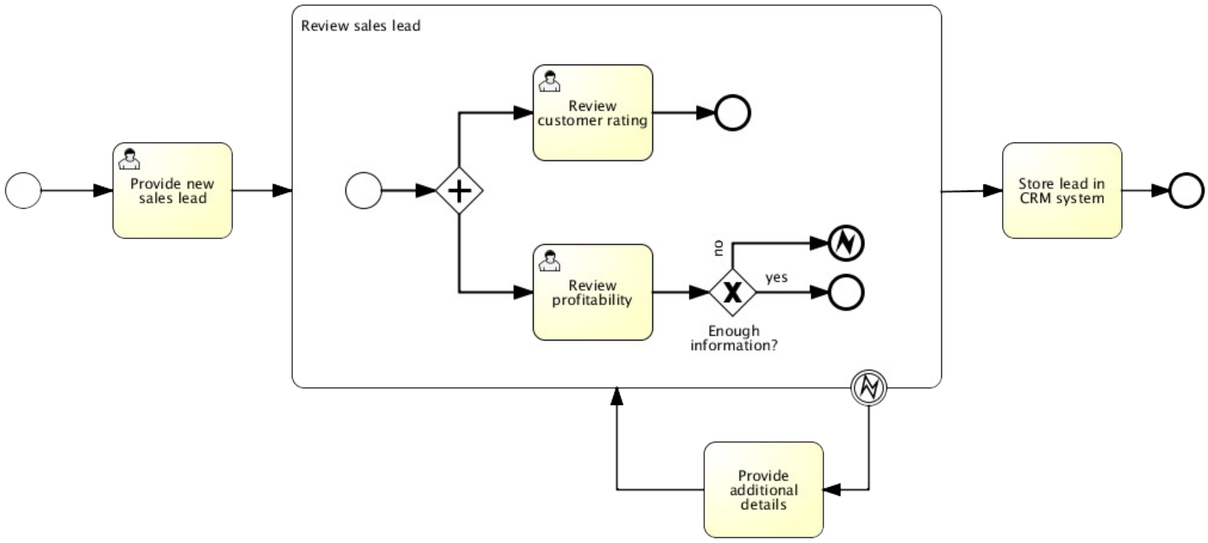
```
<error id="myError" errorCode="123" />
...
<process id="myProcess">
...
```

errorCode 用来匹配捕获的错误：

- 如果没有设置 errorRef，边界错误事件会捕获 所有错误事件，无论错误的 errorCode是什么。
- 如果设置了errorRef，并引用了一个已存的错误，边界事件就只捕获错误代码与之相同的错误。
- 如果设置了errorRef，但是BPMN 2.0中没有定义错误， errorRef 就会当做 errorCode 使用（和错误结束事件的用法类似）

Example 示例

下面的流程实例演示了如何使用错误结束事件。 当完成'审核盈利'这个用户任务是，如果没有提供足够的信息， 就会抛出错误，错误会被子流程的边界任务捕获， 所有'回顾销售'子流程中的所有节点都会销毁。 （即使'审核客户比率'还没有完成），并创建一个'提供更多信息'的用户任务。



这个流程也放在demo中了。流程XML和单元测试可以在 org.activiti.examples.bpmn.event.error 包下找到

Signal Boundary Event 信号边界事件

Description 描述

节点边界的中间捕获信号， 或简称为边界信号事件，它会捕获信号定义引用的相同信号名的信号。

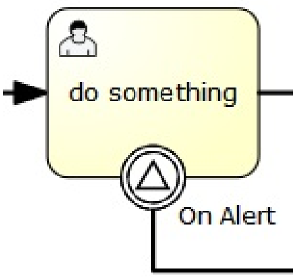
注意：与其他事件（比如边界错误事件）不同，边界信号事件不只捕获 它绑定方位的信号。

信号事件是一个全局的范围（广播语义），就是说信号可以在任何地方触发， 即便是不同的流程实例。

注意：和其他事件（比如边界错误事件）不同，捕获信号后，不会停止信号的传播。 如果你有两个信号边界事件，它们捕获相同的信号事件，两个边界事件都会被触发， 即使它们在不同的流程实例中。

Graphical notation 图形标记

边界信号事件显示为普通的中间事件（圆圈里有个小圆圈），位置在节点的边缘， 内部有一个信号小图标。信号图标是白色的（未填充）， 来表示捕获的意思。



XML representation 内容

边界信号事件定义为普通的边界事件：

```
<boundaryEvent id="boundary" attachedToRef="task" cancelActivity="true">
          <signalEventDefinition signalRef="alertSignal"/>
</boundaryEvent>
```

Example 实例

参考信号事件定义)章节

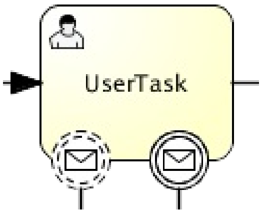
Message Boundary Event 消息边界事件

Description 描述

节点边界上的中间捕获消息， 或简称边界消息事件，根据引用的消息定义捕获相同消息名称的消息。

Graphical notation 图形标记

边界消息事件显示成一个普通的中间事件（圆圈里有个小圆圈）， 位于节点边缘， 内部是一个消息小图标。消息图标是白色（无填充）， 表示捕获语义



注意，边界消息事件可能是中断（右侧）或非中断（左侧）的。

XML representation 内容

边界消息事件定义为标准的边界事件：

```
<boundaryEvent id="boundary" attachedToRef="task" cancelActivity="true">
          <messageEventDefinition messageRef="newCustomerMessage"/>
</boundaryEvent>
```

Example 实例

参考消息事件定义章节

Cancel Boundary Event 取消边界事件

[试验]

Description 描述

在事务性子流程的边界上的中间捕获取消，或简称为边界取消事件 cancel event， 当事务取消时触发。当取消边界事件触发时，首先中断当前作用域的所有执行。 然后开始补偿事务内的所有激活的补偿边界事件。 补偿是同步执行的。例如，离开事务钱，边界事务会等待补偿执行完毕。 当补偿完成后，事务子流程会沿着取消边界事务的外出连线继续执行。

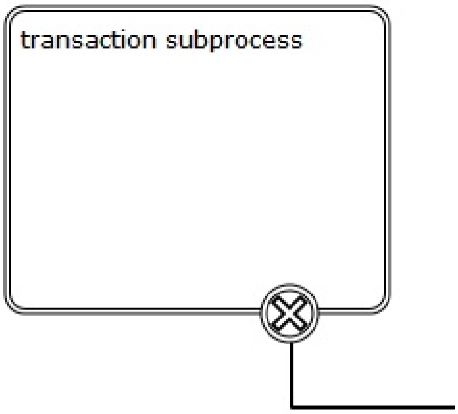
注意：每个事务子流程只能有一个取消边界事件。

注意：如果事务子流程包含内嵌子流程，补偿只会触发已经成功完成的子流程。

注意：如果取消边界子流程对应的事务子流程配置为多实例， 如果一个实例触发了取消，就会取消所有实例。

Graphical notation 图形标记

取消边界事件显示为了一个普通的中间事件（圆圈里套小圆圈），在节点的边缘，内部是一个取消小图标。取消图标是白色（无填充），表明是捕获语义。



XML representation 内容

取消边界事件定义为普通边界事件：

```
<boundaryEvent id="boundary" attachedToRef="transaction" >
          <cancelEventDefinition />
</boundaryEvent>
```

因为取消边界事件都是中断的，所以不需要使用 cancelActivity 属性。

Compensation Boundary Event 补偿边界事件

[试验]

Description 描述

节点边界的中间捕获补偿， 或简称为补偿边界事件， 可以用来设置一个节点的补偿处理器。

补偿边界事件必须使用直接引用设置唯一的补偿处理器。

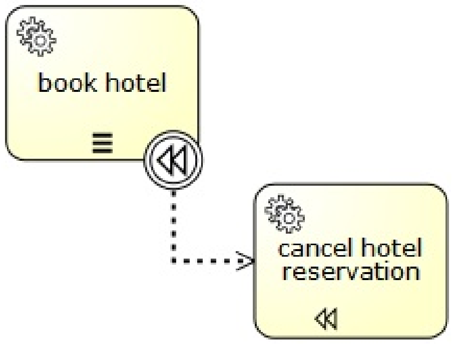
补偿边界事件与其他边界事件的策略不同。 其他边界事件（比如信号边界事件）当到达关联的节点就会被激活。 离开节点时，就会挂起，对应的事件订阅也会取消。 补偿边界事件则不同。补偿边界事件在关联的节点成功完成时激活。 当补偿事件触发或对应流程实例结束时，事件订阅才会删除。 它遵循如下规则：

- 补偿触发时，补偿边界事件对应的补偿处理器会调用相同次数，根据它对应的节点的成功次数。
- 如果补偿边界事件关联到多实例节点， 补偿事件会订阅每个实例。
- 如果补偿边界事件关联的节点中包含循环， 补偿事件会在每次节点执行时进行订阅。
- 如果流程实例结束，订阅的补偿事件都会结束。

注意：补偿边界事件不支持内嵌子流程。

Graphical notation 图形标记

补偿边界事件显示为标准中间事件（圆圈里套圆圈），位于节点边缘， 内部有一个补偿小图标。补偿图标是白色的（无填充），表示捕获语义。另外，下面的图形演示了使用无方向的关联，为边界事件设置补偿处理器。



XML representation 内容

补偿边界事件定义为标准边界事件：

```
<boundaryEvent id="compensateBookHotelEvt" attachedToRef="bookHotel" >
          <compensateEventDefinition />
</boundaryEvent>

<association associationDirection="One" id="a1"  sourceRef="compensateBookHotelEvt" targetRef="undoBookHotel" />

<serviceTask id="undoBookHotel" isForCompensation="true" activiti:class="..." />
```

因为补偿边界事件在节点成功完成后激活， 所以不支持 cancelActivity属性。

Intermediate Catching Events 中间捕获事件

所有中间捕获事件都使用同样的方式定义：

```
<intermediateCatchEvent id="myIntermediateCatchEvent" >
      <XXXEventDefinition/>
</intermediateCatchEvent>
```

中间捕获事件的定义包括

- 唯一标识（流程范围内）
- 一个结构为XXXEventDefinition的XML子元素 （比如 TimerEventDefinition等） 定义了中间捕获事件的类型。参考特定的捕获事件类型， 获得更多详情。

Timer Intermediate Catching Event 定时中间捕获事件

Description 描述

定时中间事件作为一个监听器。当执行到达捕获事件节点， 就会启动一个定时器。 当定时器触发（比如，一段时间之后），流程就会沿着定时中间事件的外出节点继续执行。

Graphical notation 图形标记

定时器中间事件显示成标准中间捕获事件，内部是一个定时器小图标



XML representation 内容

定时器中间事件定义为标准中间捕获事件。 指定类型的子元素为timerEventDefinition 元素

```
<intermediateCatchEvent id="timer">
        <timerEventDefinition>
            <timeDuration>PT5M</timeDuration>
        </timerEventDefinition>
</intermediateCatchEvent>
```

参考定时器事件定义了解配置信息。

Signal Intermediate Catching Event

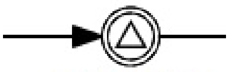
Description 描述

中间捕获信号事件 通过引用信号定义来捕获相同信号名称的信号。

注意：与其他事件（比如错误事件）不同，信号不会在捕获之后被消费。 如果你有两个激活的信号边界事件捕获相同的信号事件， 两个边界事件都会被触发， 即便它们在不同的流程实例中。

Graphical notation 图形标记

中间信号捕获事件显示为一个普通的中间事件（圆圈套圆圈）， 内部有一个信号小图标。信号小图标是白色的（无填充），表示捕获语义。



XML representation 内容

信号中间事件定义为普通的中间捕获事件。 对应类型的子元素是signalEventDefinition 元素。

```
<intermediateCatchEvent id="signal">
        <signalEventDefinition signalRef="newCustomerSignal" />
</intermediateCatchEvent>
```

Example 实例

参考信号事件定义章节。

Message Intermediate Catching Event 消息中间捕获事件

Description 描述

一个中间捕获消息事件，捕获特定名称的消息

Graphical notation 图形标记

中间捕获消息事件显示为普通中间事件（圆圈套圆圈），内部是一个消息小图标。消息图标是白色的（无填充）， 表示捕获语义。



XML representation 内容

消息中间事件定义为标准中间捕获事件。 指定类型的子元素是messageEventDefinition 元素。

```
<intermediateCatchEvent id="message">
        <messageEventDefinition signalRef="newCustomerMessage" />
</intermediateCatchEvent>
```

Example 实例

参考消息事件定义章节。

Intermediate Throwing Event 内部触发事件

所有内部触发事件的定义都是同样

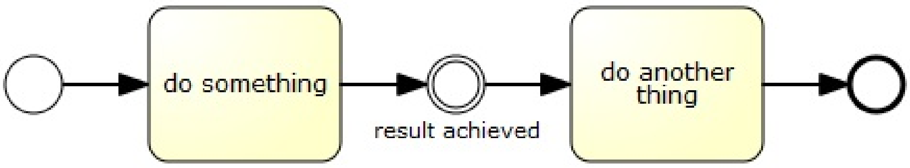
```
<intermediateThrowEvent id="myIntermediateThrowEvent" >
      <XXXEventDefinition/>
</intermediateThrowEvent>
```

内部触发事件定义包含

- 唯一标识（流程范围）
- 使用格式为XXXEventDefinition的XML子元素 （比如signalEventDefinition 等） 定义中间触发事件的类型。 参考对应触发事件的类型，了解更多信息

Intermediate Throwing None Event

下面的流程图演示了一个空中间触发事件的例子， 它通常用于表示流程中的某个状态。



通过添加执行监听器，就可以很好地监控一些 KPI。

```
<intermediateThrowEvent id="noneEvent">
  <extensionElements>
    <activiti:executionListener class="org.activiti.engine.test.bpmn.event.IntermediateNoneEventTest$MyExecutionListene
  </extensionElements>
</intermediateThrowEvent>
```

这里你可以添加自己的代码，把事件发送给BAM工具或DWH。引擎不会为这个事件做任何事情， 它直接径直通过

Signal Intermediate Throwing Event 信号中间触发事件

Description 描述

中间触发信号事件为定义的信号抛出一个信号事件。

在 activiti中，信号会广播到所有激活的处理器中（比如，所以捕获信号事件）。 信号可以通过同步和异步方式发布。

- 默认配置下，信号是同步发送的。就是说，抛出事件的流程实例会等到信号发送给所有捕获流程实例才继续执行。 捕获流程实例也会在触发流程实例的同一个事务中执行，意味着如果某个监听流程出现了技术问题（抛出异常），所有相关的实例都会失败。
- 信号也可以异步发送。这时它会在到达抛出信号事件后决定哪些处理器是激活的。 对这些激活的处理器，会保存一个异步提醒消息（任务），并发送给 jobExecutor。

Graphical notation 图形标记

中间信号触发事件显示为普通中间事件（圆圈套圆圈），内部又一个信号小图标。信号图标是黑色的（有填充），表示触发语义。



XML representation 内容

消息中间事件定义为标准中间触发事件。 指定类型的子元素是signalEventDefinition 元素。

```
<intermediateThrowEvent id="signal">
        <signalEventDefinition signalRef="newCustomerSignal" />
</intermediateThrowEvent>
```

异步信号事件如下所示：

```
<intermediateThrowEvent id="signal">
        <signalEventDefinition signalRef="newCustomerSignal" activiti:async="true" />
</intermediateThrowEvent>
```

Example 实例

参考信号事件定义章节。

Compensation Intermediate Throwing Event

[试验]

Description 描述

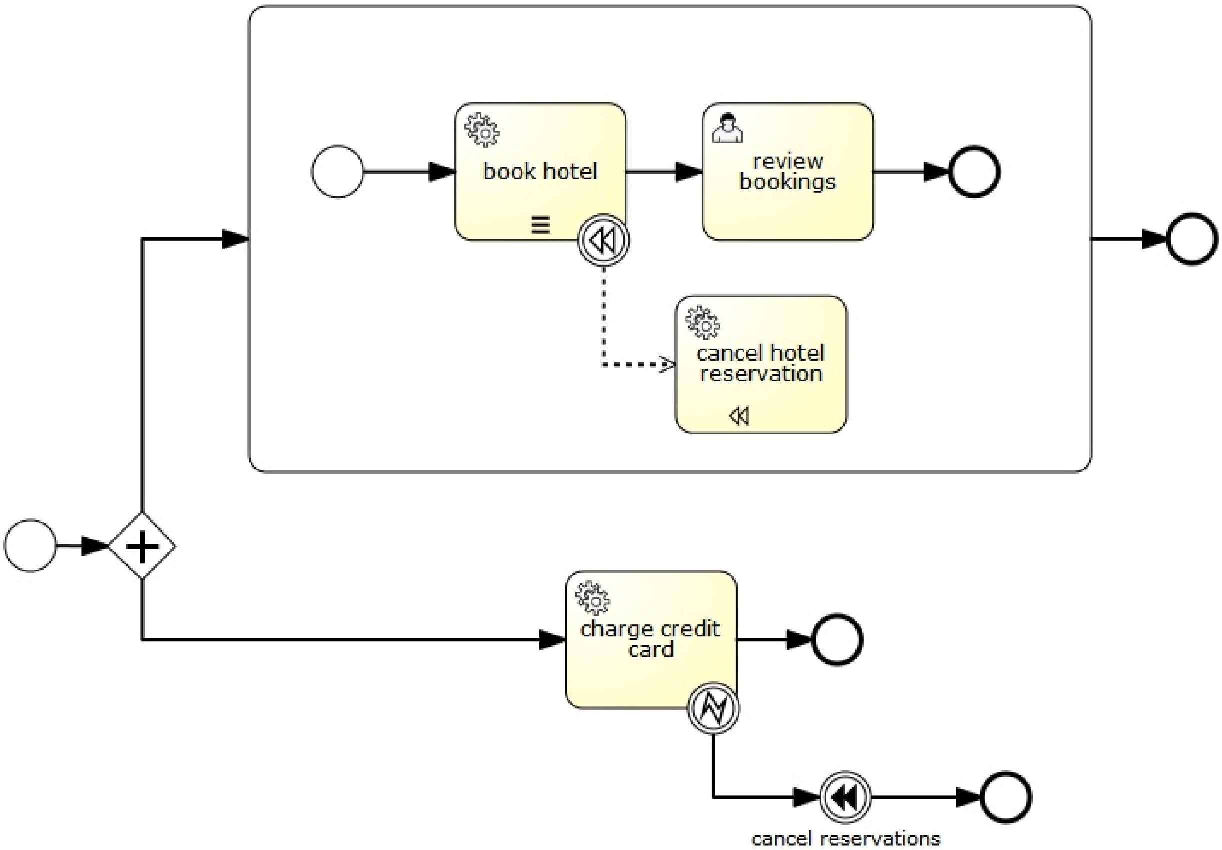
中间触发补偿事件 可以用来触发补偿。

触发补偿： 补偿可以由特定节点或包含补偿事件的作用域触发。 补偿是通过分配给节点的补偿处理器来完成的。

- 当补偿由节点触发，对应的补偿处理器会根据节点成功完成的次数执行相同次数。
- 如果补偿由当前作用域触发，当前作用域的所有节点都会执行补偿， 也包含并发分支。
- 补偿的触发是继承式的：如果执行补偿的节点是子流程，补偿会作用到子流程中包含的所有节点。 如果子流程是内嵌节点，补偿会递归触发。 然而，补偿不会传播到流程的上层： 如果补偿在子流程中触发，不会传播到子流程范围外。

bpmn规范定义，由节点触发的流程只会作用到“子流程同一级别”。
- activiti的补偿执行次序与流程执行顺序相反。 以为着最后完成的节点会最先执行补偿，诸如此类。
- 中间触发补偿事件可以用来补偿成功完成的事务性子流程。

注意： 如果补偿被一个包含子流程的作用域触发，子流程还包含了关联补偿处理器的节点，补偿只会传播到子流程，如果它已经成功完成了。 如果子流程中的节点也完成了，并关联了补偿处理器， 如果子流程包含的这些节点还没有完成，就不会执行补偿处理器。 参考下面实例：



这个流程中，我们有两个并发分支，一些分支时内嵌子流程，一个是“使用信用卡”节点。假设两个分支都启动了，第一个分支等待用户完成“审核预定”任务。第二个分支执行“使用信用卡”节点， 并发生了一个错误，这导致“取消预定”事件，并触发补偿。 这时，并发子流程还没有结束，意味着补偿事件不会传播给子流程， 所以“取消旅店预定”这个补偿处理器不会执行。 如果用户任务（就是内嵌子流程）在“取消预定”之前完成了， 补偿就会传播给内嵌子流程。

流程变量： 当补偿内嵌子流程时，用来执行补偿处理器的分支可以访问子流程的本地流程实例， 因为这时它是子流程完成的分支。 为了实现这个功能，流程变量的快照会分配给分支（为执行子流程而创建的分支）。 为此，有以下限制条件：

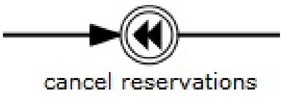
- 补偿处理器无法访问子流程内部创建的，添加到同步分支的变量。
- 分配给分支的流程变量在继承关系上层的（分配给流程实例的流程变量没有包含在快照中）： 补偿触发时，补偿处理器通过它们所在的地方访问这些流程变量。
- 变量快照只用于内嵌子流程，不适用其他节点。

已知限制：

- waitForCompletion="false"还不支持。当补偿使用中间触发补偿事件触发时， 事件没有等待， 在补偿成功结束后。
- 补偿自己由并发分支执行。并发分支的执行顺序与被补偿的节点完成次序相反。 未来 activiti 可能支持选项来顺序执行补偿。
- 补偿不会传播给callActivity调用的子流程实例。

Graphical notation 图形标记

中间补偿触发事件显示为标准中间事件（圆圈套圆圈），内部是一个补偿小图标。补偿图标是黑色的（有填充），表示触发语义。



XML representation 内容

补偿中间事件定义为普通的中间触发事件。 对应类型的子元素 是 compensateEventDefinition 元素。

```
<intermediateThrowEvent id="throwCompensation">
        <compensateEventDefinition />
</intermediateThrowEvent>
```

另外，可选参数 activityRef 可以用来触发特定作用域/节点的补偿：

```
<intermediateThrowEvent id="throwCompensation">
        <compensateEventDefinition activityRef="bookHotel" />
</intermediateThrowEvent>
```

Sequence Flow 顺序流

Description 描述

顺序流是连接两个流程节点的连线。 流程执行完一个节点后，会沿着节点的所有外出顺序流继续执行。 就是说，BPMN 2.0 默认的行为就是并发的： 两个外出顺序流会创造两个单独的，并发流程分支。

Graphical notation 图形标记

顺序流显示为从起点到终点的箭头。 箭头总是指向终点



XML representation 内容

顺序流需要流程范围内唯一的id， 以及对起点与 终点元素的引用。

```
<sequenceFlow id="flow1" sourceRef="theStart" targetRef="theTask" />
```

Conditional sequence flow 条件顺序流

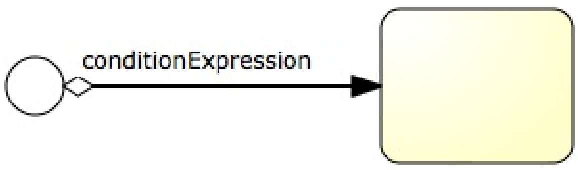
Description 描述

可以为顺序流定义一个条件。离开一个BPMN 2.0节点时， 默认会计算外出顺序流的条件。如果条件结果为 true, 就会选择外出顺序流继续执行。当多条顺序流被选中时， 就会创建多条分支， 流程会继续以并行方式继续执行。

注意：上面的讨论仅涉及 BPMN 2.0 节点（和事件）， 不包括网关。网关会用特定的方式处理顺序流中的条件， 这与网关类型相关。

Graphical notation 图形标记

条件顺序流显示为一个正常的顺序流，不过在起点有一个菱形。 条件表达式也会显示在顺序流上。



XML representation 内容

条件顺序流定义为一个正常的顺序流， 包含 conditionExpression 子元素。 注意目前只支持 tFormalExpressions， 如果没有设置xsi:type="" , 就会默认值支持目前支持的表达式类型

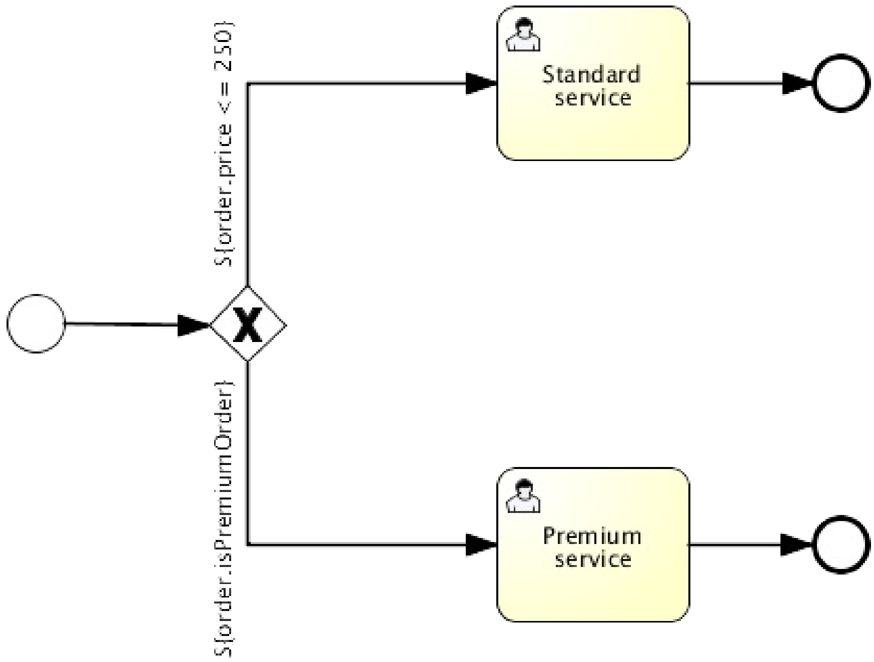
```
<sequenceFlow id="flow" sourceRef="theStart" targetRef="theTask">
  <conditionExpression xsi:type="tFormalExpression">
    <![CDATA[${order.price > 100 && order.price < 250}]]>
```

```
    </conditionExpression>
  </sequenceFlow>
```

当前条件表达式只能使用 **UEL**， 可以参考表达式章节获取更多信息。 使用的表达式需要返回 boolean 值，否则会在解析表达式时抛出异常。

- 下面的例子引用了流程变量的数据， 通过 getter 调用 JavaBean。
- 这个例子通过调用方法返回一个 boolean 值。

在 activiti 发布包中，包含以下流程实例，使用了值和方法表达式 （参 考 org.activiti.examples.bpmn.expression)包）：



Default sequence flow 默认顺序流

Description 描述

所有的 BPMN 2.0 任务和网关都可以设置一个默认顺序流。 只有在节点的其他外出顺序流不能被选中是，才会使用它作为外出顺序流继续执行。 默认顺序流的条件设置不会生效。

Graphical notation 图形标记

默认顺序流显示为了普通顺序流，起点有一个“斜线”标记。

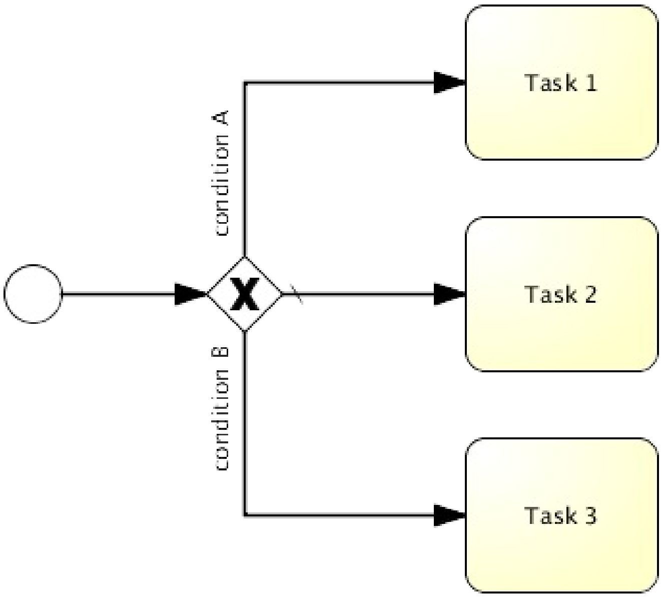


XML representation 内容

默认顺序流通过对应节点的 default 属性定义。 下面的 XML 代码演示了排他网关设置了默认顺序流 flow 2 。 只有当 conditionA 和conditionB 都返回 false 时， 才会选择它作为外出连线继续执行。

```
<exclusiveGateway id="exclusiveGw" name="Exclusive Gateway" default="flow2" />
<sequenceFlow id="flow1" sourceRef="exclusiveGw" targetRef="task1">
  <conditionExpression xsi:type="tFormalExpression">${conditionA}</conditionExpression>
</sequenceFlow>
<sequenceFlow id="flow2" sourceRef="exclusiveGw" targetRef="task2"/>
<sequenceFlow id="flow3" sourceRef="exclusiveGw" targetRef="task3">
  <conditionExpression xsi:type="tFormalExpression">${conditionB}</conditionExpression>
</sequenceFlow>
```

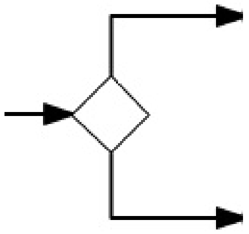
对应下面的图形显示：



Gateways 网关

网关用来控制流程的流向（或 像BPMN 2.0 里描述的那样，流程的tokens。） 网关可以消费也可以生成 token。

网关显示成菱形图形，内部有有一个小图标。 图标表示网关的类型



Exclusive Gateway 排他网关

Description 描述

排他网关（也叫异或（XOR）网关，或更技术性的叫法 基于数据的排他网关）， 用来在流程中实现决策。 当流程执行到这个网关，所有外出顺序流都会被处理一遍。 其中条件解析为true的顺序流（或者没有设置条件，概念上在顺序流上定义了一个'true'） 会被选中，让流程继续运行。

注意这里的外出顺序流 与 **BPMN 2.0** 通常的概念是不同的。通常情况下，所有条件结果为**true**的顺序流 都会被选中，以并行方式执行，但排他网关只会选择一条顺序流执行。 就是说，虽然多个顺序流的条件结果为**true**， 那么**XML**中的第一个顺序流（也只有这一条）会被选中，并用来继续运行流程。 如果没有选中任何顺序流，会抛出一个异常

Graphical notation 图形标记

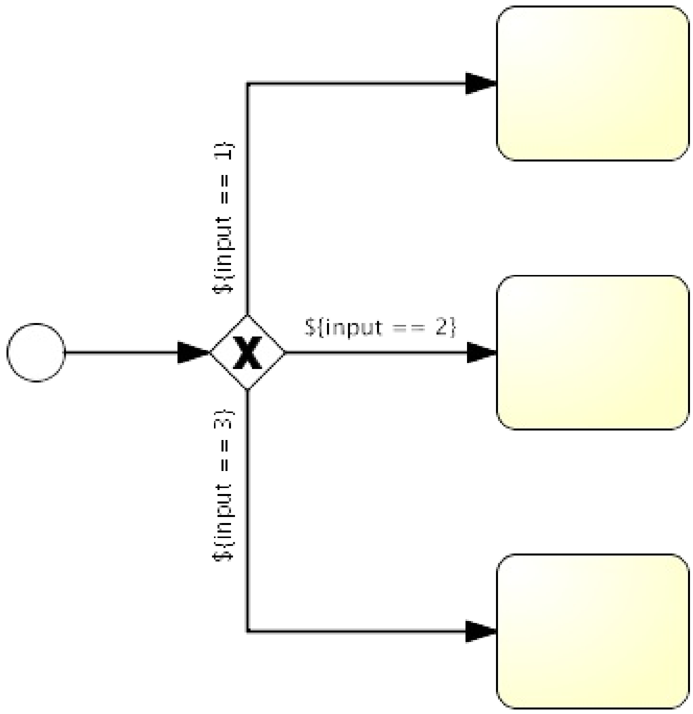
排他网关显示成一个普通网关（比如，菱形图形）， 内部是一个“X”图标，表示异或（XOR）语义。 注意，没有内部图标的网关，默认为排他网关。 BPMN 2.0 规范不允许在同一个流程定义中同时使用没有X和有X的菱形图形。



XML representation 内容

排他网关的XML内容是很直接的：用一行定义了网关，条件表达式定义在外出顺序流中。 参考条件顺序流 获得这些表达式的可用配置。

参考下面模型实例：



它对应的XML内容如下：

```
<exclusiveGateway id="exclusiveGw" name="Exclusive Gateway" />

<sequenceFlow id="flow2" sourceRef="exclusiveGw" targetRef="theTask1">
  <conditionExpression xsi:type="tFormalExpression">${input == 1}</conditionExpression>
</sequenceFlow>

<sequenceFlow id="flow3" sourceRef="exclusiveGw" targetRef="theTask2">
  <conditionExpression xsi:type="tFormalExpression">${input == 2}</conditionExpression>
</sequenceFlow>

<sequenceFlow id="flow4" sourceRef="exclusiveGw" targetRef="theTask3">
  <conditionExpression xsi:type="tFormalExpression">${input == 3}</conditionExpression>
</sequenceFlow>
```

Parallel Gateway 并行网关

Description 描述

网关也可以表示流程中的并发情况。最简单的并发网关是 并行网关，它允许将流程分成多条分支，也可以把多条分支 汇聚到一起。

并行网关的功能是基于进入和外出的顺序流的：

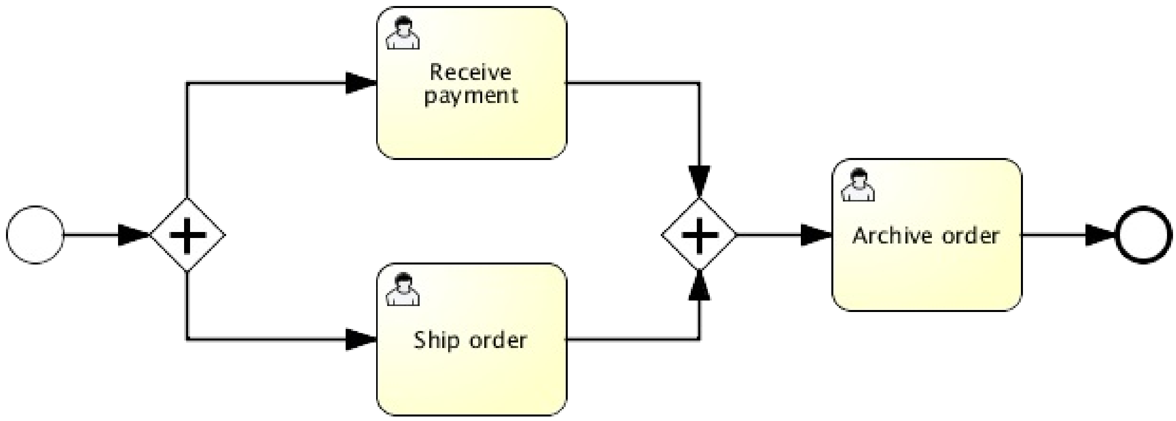
- 分支： 并行后的所有外出顺序流，为每个顺序流都创建一个并发分支。
- 汇聚： 所有到达并行网关，在此等待的进入分支， 直到所有进入顺序流的分支都到达以后， 流程就会通过汇聚网关。

注意，如果同一个并行网关有多个进入和多个外出顺序流， 它就同时具有分支和汇聚功能。这时，网关会先汇聚所有进入的顺序流，然后再切分成多个并行分支。

与其他网关的主要区别是，并行网关不会解析条件。 即使顺序流中定义了条件，也会被忽略。

Graphical notation 图形标记

并行网关显示成一个普通网关（菱形）内部是一个“加号”图标， 表示“与（AND）”语义。



XML representation 内容

定义并行网关只需要一行 XML：

```
<parallelGateway id="myParallelGateway" />
```

实际发生的行为（分支，聚合，同时分支聚合）， 要根据并行网关的顺序流来决定。

参考如下代码：

```
<startEvent id="theStart" />
<sequenceFlow id="flow1" sourceRef="theStart" targetRef="fork" />

<parallelGateway id="fork" />
<sequenceFlow sourceRef="fork" targetRef="receivePayment" />
<sequenceFlow sourceRef="fork" targetRef="shipOrder" />

<userTask id="receivePayment" name="Receive Payment" />
<sequenceFlow sourceRef="receivePayment" targetRef="join" />

<userTask id="shipOrder" name="Ship Order" />
<sequenceFlow sourceRef="shipOrder" targetRef="join" />

<parallelGateway id="join" />
<sequenceFlow sourceRef="join" targetRef="archiveOrder" />

<userTask id="archiveOrder" name="Archive Order" />
<sequenceFlow sourceRef="archiveOrder" targetRef="theEnd" />

<endEvent id="theEnd" />
```

上面例子中，流程启动之后，会创建两个任务：

```
ProcessInstance pi = runtimeService.startProcessInstanceByKey("forkJoin");
TaskQuery query = taskService.createTaskQuery()
                         .processInstanceId(pi.getId())
                         .orderByTaskName()
                         .asc();

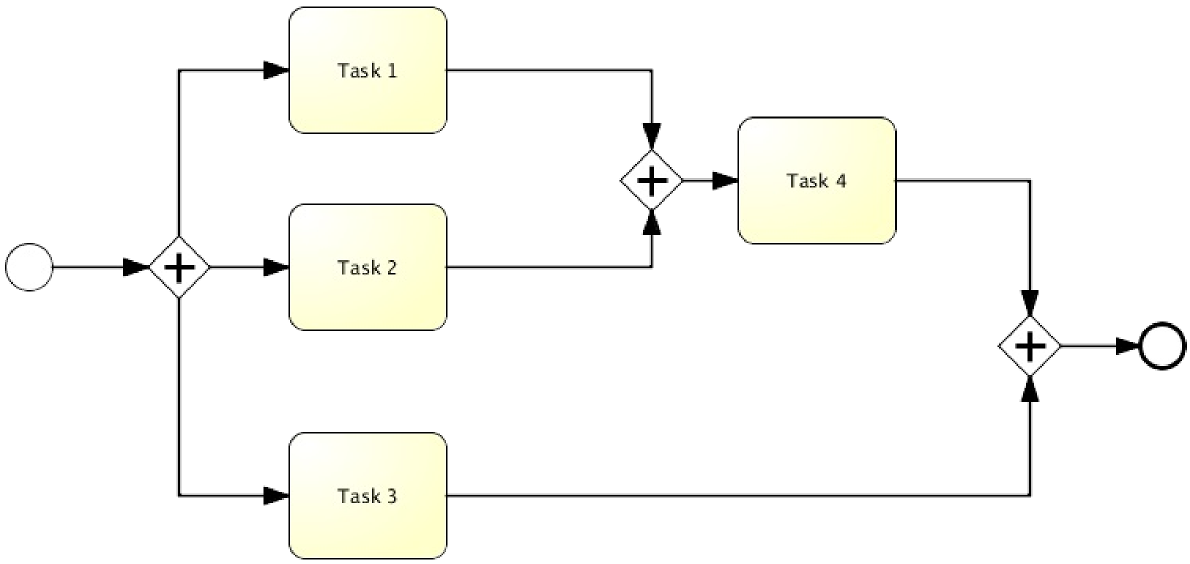
List<Task> tasks = query.list();
```

```
assertEquals(2, tasks.size());

Task task1 = tasks.get(0);
assertEquals("Receive Payment", task1.getName());
Task task2 = tasks.get(1);
assertEquals("Ship Order", task2.getName());
```

当两个任务都完成时，第二个并行网关会汇聚两个分支，因为它只有一条外出连线， 不会创建并行分支， 只会创建归档订单任务。

注意并行网关不需要是“平衡的”（比如， 对应并行网关的进入和外出节点数目相等）。并行网关只是等待所有进入顺序流，并为每个外出顺序流创建并发分支， 不会受到其他流程节点的影响。 所以下面的流程在 BPMN 2.0中是合法的：



Inclusive Gateway 包含网关

Description 描述

包含网关可以看做是排他网关和并行网关的结合体。 和排他网关一样，你可以在外出顺序流上定义条件，包含网关会解析它们。 但是主要的区别是包含网关可以选择多于一条顺序流， 这和并行网关一样。

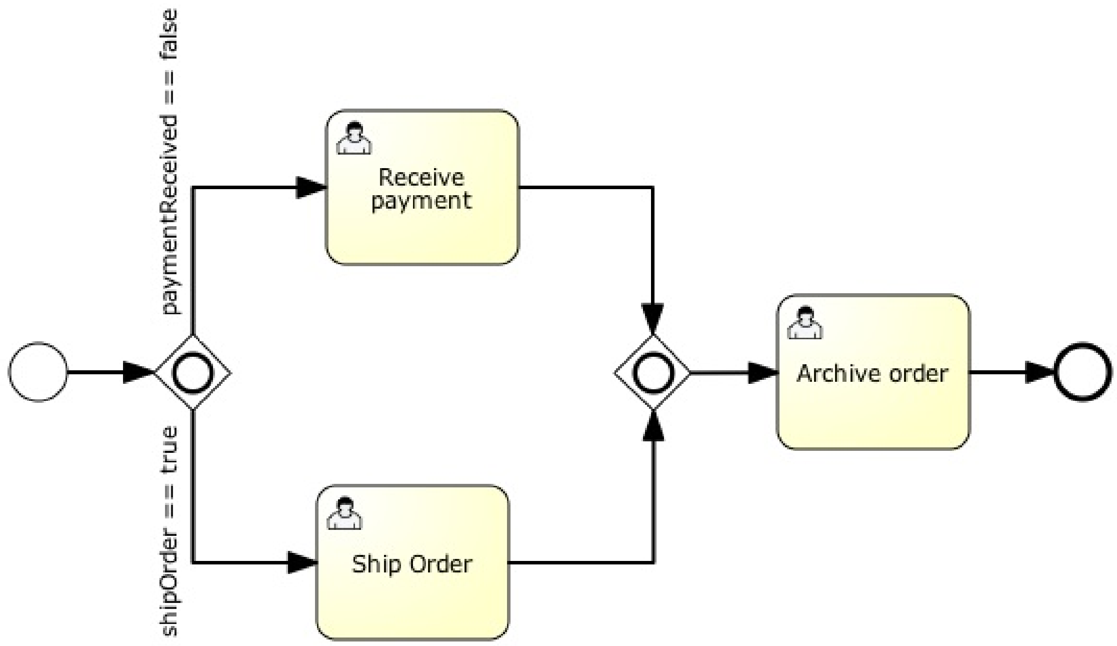
包含网关的功能是基于进入和外出顺序流的：

- 分支： 所有外出顺序流的条件都会被解析，结果为true的顺序流会以并行方式继续执行，会为每个顺序流创建一个分支。
- 汇聚： 所有并行分支到达包含网关，会进入等待章台， 直到每个包含流程token的进入顺序流的分支都到达。 这是与并行网关的最大不同。换句话说，包含网关只会等待被选中执行了的进入顺序流。 在汇聚之后，流程会穿过包含网关继续执行。

注意，如果同一个包含节点拥有多个进入和外出顺序流， 它就会同时含有分支和汇聚功能。这时，网关会先汇聚所有拥有流程token的进入顺序流， 再根据条件判断结果为true的外出顺序流，为它们生成多条并行分支。

Graphical notation 图形标记

并行网关显示为一个普通网关（菱形），内部包含一个圆圈图标。



XML representation 内容

定义一个包含网关需要一行 XML：

```
<inclusiveGateway id="myInclusiveGateway" />
```

实际的行为（分支，汇聚或同时分支汇聚），是由连接在包含网关的顺序流决定的。

参考如下代码：

```
<startEvent id="theStart" />
<sequenceFlow id="flow1" sourceRef="theStart" targetRef="fork" />

<inclusiveGateway id="fork" />
<sequenceFlow sourceRef="fork" targetRef="receivePayment" >
<conditionExpression xsi:type="tFormalExpression">${paymentReceived == false}</conditionExpression>
</sequenceFlow>
<sequenceFlow sourceRef="fork" targetRef="shipOrder" >
<conditionExpression xsi:type="tFormalExpression">${shipOrder == true}</conditionExpression>
</sequenceFlow>

<userTask id="receivePayment" name="Receive Payment" />
<sequenceFlow sourceRef="receivePayment" targetRef="join" />

<userTask id="shipOrder" name="Ship Order" />
<sequenceFlow sourceRef="shipOrder" targetRef="join" />

<inclusiveGateway id="join" />
<sequenceFlow sourceRef="join" targetRef="archiveOrder" />

<userTask id="archiveOrder" name="Archive Order" />
<sequenceFlow sourceRef="archiveOrder" targetRef="theEnd" />

<endEvent id="theEnd" />
```

在上面的例子中，流程开始之后，如果流程变量为 paymentReceived == false 和 shipOrder == true， 就会创建两个任务。如果，只有一个流程变量为true，就会只创建一个任务。 如果没有条件为true，就会抛出一个异常。 如果想避免异常，可以定义一个默认顺序流。下面的例子中，会创建一个任务，发货任务：

```
HashMap<String, Object> variableMap = new HashMap<String, Object>();
          variableMap.put("receivedPayment", true);
          variableMap.put("shipOrder", true);
          ProcessInstance pi = runtimeService.startProcessInstanceByKey("forkJoin");
TaskQuery query = taskService.createTaskQuery()
                         .processInstanceId(pi.getId())
                         .orderByTaskName()
                         .asc();

List<Task> tasks = query.list();
assertEquals(1, tasks.size());

Task task = tasks.get(0);
assertEquals("Ship Order", task.getName());
```

当任务完成后，第二个包含网关会汇聚两个分支， 因为只有一个外出顺序流，所以不会创建并行分支， 只有归档订单任务会被激活。

注意，包含网关不需要“平衡”（比如，对应包含网关的进入和外出数目需要相等）。包含网关会等待所有进入顺序流完成，并为每个外出顺序流创建并行分支，不会受到流程中其他元素的影响。

Event-based Gateway 基于事件网关

Description 描述

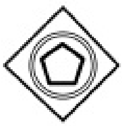
基于事件网关允许根据事件判断流向。网关的每个外出顺序流都要连接到一个中间捕获事件。 当流程到达一个基于事件网关，网关会进入等待状态：会暂停执行。 与此同时，会为每个外出顺序流创建相对的事件订阅。

注意基于事件网关的外出顺序流和普通顺序流不同。这些顺序流不会真的"执行"。 相反，它们让流程引擎去决定执行到基于事件网关的流程需要订阅哪些事件。 要考虑以下条件：

- 基于事件网关必须有两条或以上外出顺序流。
- 基于事件网关后，只能使用 intermediateCatchEvent 类型。 （activiti 不支持基于事件网关后连接 ReceiveTask。）
- 连接到基于事件网关的 intermediateCatchEvent 只能有一条进入顺序流。

Graphical notation 图形标记

基于事件网关和其他BPMN网关一样显示成一个菱形， 内部包含指定图标。

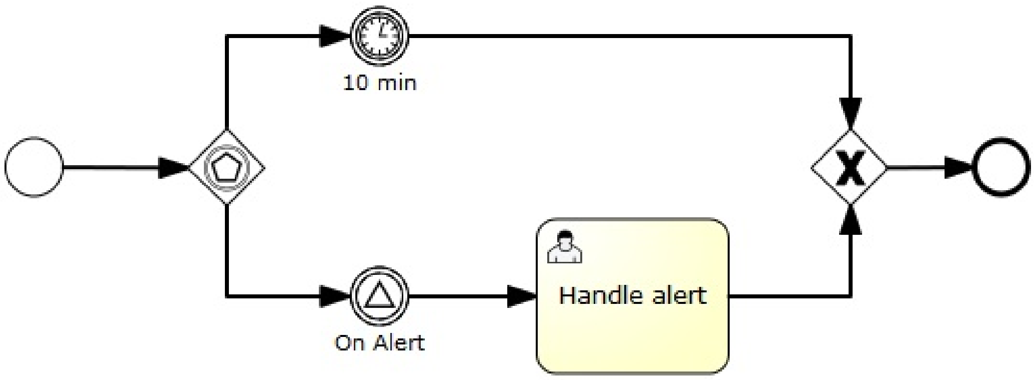


XML representation 内容

用来定义基于事件网关的XML元素是 eventBasedGateway。

Example(s) 实例

下面的流程是一个使用基于事件网关的例子。当流程执行到基于事件网关时， 流程会暂停执行。与此同时，流程实例会订阅警告信号事件，并创建一个10分钟后触发的定时器。 这会产生流程引擎为一个信号事件等待10分钟的效果。如果10分钟内发出信号，定时器就会取消，流程会沿着信号执行。 如果信号没有出现，流程会沿着定时器的方向前进，信号订阅会被取消。




```
<definitions id="definitions"
        xmlns="http://www.omg.org/spec/BPMN/20100524/MODEL"
        xmlns:activiti="http://activiti.org/bpmn"
        targetNamespace="Examples">

        <signal id="alertSignal" name="alert" />

        <process id="catchSignal">

                <startEvent id="start" />

                <sequenceFlow sourceRef="start" targetRef="gw1" />

                <eventBasedGateway id="gw1" />

                <sequenceFlow sourceRef="gw1" targetRef="signalEvent" />
                <sequenceFlow sourceRef="gw1" targetRef="timerEvent" />

                <intermediateCatchEvent id="signalEvent" name="Alert">
                        <signalEventDefinition signalRef="alertSignal" />
                </intermediateCatchEvent>

                <intermediateCatchEvent id="timerEvent" name="Alert">
                        <timerEventDefinition>
                                <timeDuration>PT10M</timeDuration>
                        </timerEventDefinition>
                </intermediateCatchEvent>

                <sequenceFlow sourceRef="timerEvent" targetRef="exGw1" />
                <sequenceFlow sourceRef="signalEvent" targetRef="task" />

                <userTask id="task" name="Handle alert"/>

                <exclusiveGateway id="exGw1" />

                <sequenceFlow sourceRef="task" targetRef="exGw1" />
                <sequenceFlow sourceRef="exGw1" targetRef="end" />

                <endEvent id="end" />
</process>
</definitions>
```

Tasks 任务

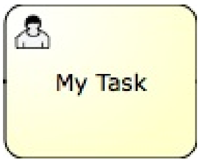
User Task 用户任务

Description 描述

用户任务用来设置必须由人员完成的工作。当流程执行到用户任务，会创建一个新任务，并把这个新任务加入到分配人或群组的任务列表中。

Graphical notation 图形标记

用户任务显示成一个普通任务（圆角矩形），左上角有一个小用户图标



XML representation 内容

XML 中的用户任务定义如下。id 属性是必须的。name 属性是可选的。

```
<userTask id="theTask" name="Important task" />
```

用户任务也可以设置描述。实际上所有 BPMN 2.0 元素 都可以设置描述。 添加 documentation 元素可以定义描述。

```
<userTask id="theTask" name="Schedule meeting" >
  <documentation>
          Schedule an engineering meeting for next week with the new hire.
  </documentation>
```

描述文本可以通过标准的java方法来获得:

```
task.getDescription()
```

Due Date 持续时间

任务可以用一个字段来描述任务的持续时间。可以使用查询 API 来对持续时间进行搜索， 根据在时间之前或之后进行搜索。

我们提供了一个节点扩展，在任务定义中设置一个表达式，这样在任务创建时就可以为它设置初始持续时间。表达式应该是 java.util.Date， java.util.String (ISO8601格式)，ISO8601 持续时间 (比如 PT50M)或null。 例如：你可以在流程中使用上述格式输入日期，或在前一个服务任务中计算一个时间。 这里使用了持续时间，持续时间会基于当前时间进行计算，再通过给定的时间段累加。 比如，使用"PT30M"作为持续时间，任务就会从现在开始持续 30 分钟。

```
<userTask id="theTask" name="Important task" activiti:dueDate="${dateVariable}"/>
```

User assignment 用户分配

用户任务可以直接分配给一个用户。 这可以通过 humanPerformer 元素定义。humanPerformer 定义需要一个 resourceAssignmentExpression 来实际定义用户。 当前，只支持 formalExpressions。

```
<process ... >

  ...

  <userTask id='theTask' name='important task' >
    <humanPerformer>
      <resourceAssignmentExpression>
        <formalExpression>kermit</formalExpression>
      </resourceAssignmentExpression>
    </humanPerformer>
  </userTask>
```

只有一个用户可以坐拥任务的执行者分配给用户。 在 activiti 中，用户叫做执行者。 拥有执行者的用户不会出现在其他人的任务列表中， 只能出现执行者的个人任务列表中。

直接分配给用户的任务可以通过 TaskService 像下面这样获取：

```
List<Task> tasks = taskService.createTaskQuery().taskAssignee("kermit").list();
```

任务也可以加入到人员的候选任务列表中。 这时，需要使用potentialOwner 元素。 用法和 humanPerformer 元素类似。注意它需要指定表达式中的每个项目是人员还是群组（引擎猜不出来）。

```
<process ... >

  ...

  <userTask id='theTask' name='important task' >
    <potentialOwner>
      <resourceAssignmentExpression>
        <formalExpression>user(kermit), group(management)</formalExpression>
      </resourceAssignmentExpression>
    </potentialOwner>
  </userTask>
```

使用 potential owner 元素定义的任务，可以像下面这样获取 （使用TaskQuery 的发那个发与查询设置了执行者的任务类似）：

```
List<Task> tasks = taskService.createTaskQuery().taskCandidateUser("kermit");
```

这会获取所有kermit为候选人的任务， 例如：表达式中包含 user(kermit)。 这也会获得所有分配包含 kermit 这个成员的群组（比如，group(management)，前提是 kermit 是这个组的成员， 并且使用了activiti 的账号组件）。 用户所在的群组是在运行阶段获取的， 它们可 以通过 IdentityService 进行管理。

如果没有显示指定设置的是用户还是群组，引擎会默认当做群组处理。所以下面的设置与使用 group(accountancy) 效果一样。

```
<formalExpression>accountancy</formalExpression>
```

Activiti extensions for task assignment 对任务分配的扩展

当分配不复杂时，用户和组的设置非常麻烦。 为避免复杂性，可以使用用户任务的自定义扩展。

- assignee属性：这个自定义扩展可以直接把用户任务分配给指定用户。

它和使用上面定义的 humanPerformer 效果完全一样。

- candidateUsers 属性：这个自定义扩展可以为任务设置候选人。

它和使用上面定义的 potentialOwner 效果完全一样。 注意它不需要像使用potential owner 通过 user(kermit)声明， 因为这个属性只能用于人员。

- candidateGroups 属性：这个自定义扩展可以为任务设置候选组。

它和使用上面定义的 potentialOwner 效果完全一样。 注意它不需要像使用 potentialOwner 通过 group(management) 声明，因为这个属性只能用于群组。

- candidateUsers 和 candidateGroups 可以同时设置在同一个用户任务中。

注意：虽然 activiti 提供了一个账号管理组件， 也提供了IdentityService， 但是账号组件不会检测设置的用户是否村爱。 它嵌入到应用中，也允许 activiti 与其他已存的账户管理方 案集成。

Custom identity link types (Experimental) 自定义身份链接类型(试验)

[试验]

BPMN 标准支持一个指定的用户或 humanperformer 或一组用户，形成一个潜在的池 potentialowners 为用户分配的定义。此外，Activiti 定义扩展属性的元素，可以代表任务受让人或候选人的所有者的用户任务。

支持的身份链接类型有：

```
public class IdentityLinkType {
  /* Activiti native roles */
  public static final String ASSIGNEE = "assignee";
  public static final String CANDIDATE = "candidate";
  public static final String OWNER = "owner";
  public static final String STARTER = "starter";
  public static final String PARTICIPANT = "participant";
}
```

BPMN 标准和 Activiti 实例授权身份是 user（用户）和 group（组）。如前所述，Activiti 身份管理的实施不是用于生产使用，但应扩展取决于支持授权方案。

如果额外的链接类型是必需的，自定义资源可以被定义为与下面的语法扩展元素：

```
<userTask id="theTask" name="make profit">
  <extensionElements>
    <activiti:customResource activiti:name="businessAdministrator">
      <resourceAssignmentExpression>
        <formalExpression>user(kermit), group(management)</formalExpression>
      </resourceAssignmentExpression>
    </activiti:customResource>
  </extensionElements>
</userTask>
```

自定义链接表达式添加到 TaskDefinition 类：

```
  protected Map<String, Set<Expression>> customUserIdentityLinkExpressions =
      new HashMap<String, Set<Expression>>();
  protected Map<String, Set<Expression>> customGroupIdentityLinkExpressions =
      new HashMap<String, Set<Expression>>();
```

```
  public Map<String,
         Set<Expression>> getCustomUserIdentityLinkExpressions() {
    return customUserIdentityLinkExpressions;
  }

  public void addCustomUserIdentityLinkExpression(String identityLinkType,
                                                  Set<Expression> idList) {
    customUserIdentityLinkExpressions.put(identityLinkType, idList);
  }

  public Map<String,
         Set<Expression>> getCustomGroupIdentityLinkExpressions() {
    return customGroupIdentityLinkExpressions;
  }

  public void addCustomGroupIdentityLinkExpression(String identityLinkType,
                                                   Set<Expression> idList) {
    customGroupIdentityLinkExpressions.put(identityLinkType, idList);
  }
```

这是由在运行时的 UserTaskActivityBehavior handleAssignments 方法落实的。

最后，该 IdentityLinkType 类必须扩展支持自定义身份链接类型：

```
package com.yourco.engine.task;

public class IdentityLinkType
    extends org.activiti.engine.task.IdentityLinkType
{
    public static final String ADMINISTRATOR = "administrator";

    public static final String EXCLUDED_OWNER = "excludedOwner";
}
```

这是一个数据库实体，包含了自定义身份链接类型 administrator 与一个单一的用户 (gonzo) 和组 (engineering)。

SELECT * FROM ACT_RU_IDENTITYLINK;

ID_	REV_	GROUP_ID_	TYPE_	USER_ID_	TASK_ID_	PROC_INST_ID_	PROC_DEF_ID_
8991	1	*null*	starter	kermit	*null*	8989	*null*
8995	1	*null*	administrator	gonzo	8994	*null*	*null*
8996	1	*null*	participant	gonzo	*null*	8989	*null*
8997	1	engineering	administrator	*null*	8994	*null*	*null*

Custom Assignment via task listeners 通过任务监听器自定义分配任务

如果上面的方式还不满足需求，可以使用任务监听器在创建事件委托自定义任务逻辑：

```
<userTask id="task1" name="My task" >
  <extensionElements>
    <activiti:taskListener event="create" class="org.activiti.MyAssignmentHandler" />
  </extensionElements>
</userTask>
```

该 DelegateTask， 传递到 TaskListener 的实现，允许设置的受让人和候选的用户或组：

```
public class MyAssignmentHandler implements TaskListener {

  public void notify(DelegateTask delegateTask) {
    // Execute custom identity lookups here
```

```
    // and then for example call following methods:
    delegateTask.setAssignee("kermit");
    delegateTask.addCandidateUser("fozzie");
    delegateTask.addCandidateGroup("management");
    ...
  }

}
```

使用 spring 时，可以使用向上面章节中介绍的自定义分配属性，使用表达式 把任务监听器设置为 spring 代理的 bean， 让这个监听器监听任务的创建事件。 下面的例子中，执行者会通过调用 ldapService 这个spring bean 的 findManagerOfEmployee 方法获得。 流程变量 emp 会作为参数传递给 bean。

```
<userTask id="task" name="My Task" activiti:assignee="${ldapService.findManagerForEmployee(emp)}"/>
```

也可以用来设置候选人和候选组：

```
<userTask id="task" name="My Task" activiti:candidateUsers="${ldapService.findAllSales()}"/>
```

注意方法返回类型只能为 String 或 Collection （对应候选人和候选组）：

```
public class FakeLdapService {

  public String findManagerForEmployee(String employee) {
    return "Kermit The Frog";
  }

  public List<String> findAllSales() {
    return Arrays.asList("kermit", "gonzo", "fozzie");
  }

}
```

Script Task 脚本任务

Description 描述

脚本任务时一个自动节点。当流程到达脚本任务， 会执行对应的脚本。

Graphical notation 图形标记

脚本任务显示为标准 BPMN 2.0 任务（圆角矩形）， 左上角有一个脚本小图标。



XML representation 内容

脚本任务定义需要指定 script 和 scriptFormat。

```
<scriptTask id="theScriptTask" name="Execute script" scriptFormat="groovy">
  <script>
    sum = 0
    for ( i in inputArray ) {
      sum += i
    }
  </script>
</scriptTask>
```

scriptFormat 的值必须兼容 JSR-223（java 平台的脚本语言）。默认 Javascript 会包含在 JDK 中，不需要额外的依赖。 如果你想使用其他（JSR-223兼容）的脚本引擎， 需要把对应的 jar 添加到 classpath 下，并使用合适的名称。比如，activiti 单元测试经常使用 groovy， 因为语法比 java 简单太多。(译者注：新出的项目管理工具 Gradle 也是使用了 groovy,详见《Gradle 2 用户指南》)

注意，groovy 脚本引擎放在 groovy-all.jar中。在 2.0 版本之前， 脚本引擎是 Groovy jar 的一部分。这样，需要添加如下依赖：

```
<dependency>
      <groupId>org.codehaus.groovy</groupId>
      <artifactId>groovy-all</artifactId>
      <version>2.x.x<version>
</dependency>
```

脚本中的变量

到达脚本任务的流程可以访问的所有流程变量，都可以在脚本中使用。实例中，脚本变量'inputArray'其实是流程变量（整数数组）。

```
<script>
    sum = 0
    for ( i in inputArray ) {
      sum += i
    }
</script>
```

也可以在脚本中设置流程变量，直接调用 execution.setVariable("variableName",variableValue)。 默认，不会自动保存变量（注意 ：activiti 5.12 之前存在这个问题）。 可以在脚本中自动保存任何变量。 （比如上例中的 sum ），只要把 scriptTask 的autoStoreVariables 属性设置为 true。 然而，最佳实践是不要用它，而是显示调用 execution.setVariable()， 因为一些当前版本的 JDK对于一些脚本语言，无法实现自动保存变量。 参考这里 获得更多信息。

```
<scriptTask id="script" scriptFormat="JavaScript" activiti:autoStoreVariables="false">
```

参数默认为 false，意思是如果没有为脚本任务定义设置参数， 所有声明的变量将只存在于脚本执行的阶段。

如何在脚本中设置变量的例子：

```
<script>
    def scriptVar = "test123"
    execution.setVariable("myVar", scriptVar)
</script>
```

注意：下面这些命名已被占用，不能用作变量名： : out, out:print, lang:import, context, elcontext

Script results 脚本结果

脚本任务的返回值可以通过制定流程变量的名称，分配给已存或一个新流程变量， 使用脚本任务定义的 'activiti:resultVariable' 属性。 任何已存的流程变量都会被脚本执行的结果覆盖。 如果没有指定返回变量名，脚本的返回值会被忽略。

```
<scriptTask id="theScriptTask" name="Execute script" scriptFormat="juel" activiti:resultVariable="myVar">
  <script>#{echo}</script>
</scriptTask>
```

上例中，脚本的结果（表达式'#{echo}'的值） 在脚本完成后，会设置到'myVar'变量中。

Java Service Task 服务任务

Description 描述

Java 服务任务用来调用外部 Java 类

Graphical notation 图形标记

服务任务显示为圆角矩形，左上角有一个齿轮小图标



XML representation 内容

有4钟方法来声明 java 调用逻辑：

- 实现 JavaDelegate 或 ActivityBehavior
- 执行解析代理对象的表达式
- 调用一个方法表达式
- 调用一直值表达式

执行一个在流程执行中调用的类， 需要在'activiti:class'属性中设置全类名。

```
<serviceTask id="javaService"
             name="My Java Service Task"
             activiti:class="org.activiti.MyJavaDelegate" />
```

参考实现章节 了解更多使用类的信息

也可以使用表达式调用一个对象。对象必须遵循一些规则， 并使用activiti:class 属性进行创建。（了解更多）。

```
<serviceTask id="serviceTask" activiti:delegateExpression="${delegateExpressionBean}" />
```

这里，delegateExpressionBean 是一个实现了 JavaDelegate 接口的bean， 它定义在实例的 spring 容器中。

要指定执行的 UEL 方法表达式， 需要使用 activiti:expression。

```
<serviceTask id="javaService"
         name="My Java Service Task"
         activiti:expression="#{printer.printMessage()}" />
```

方法 printMessage（无参数）会调用 名为 printer 对象的方法。 也可以为表达式中的方法传递参数。

```
<serviceTask id="javaService"
         name="My Java Service Task"
         activiti:expression="#{printer.printMessage(execution, myVar)}" />
```

这会调用名为 printer 对象上的方法 printMessage。 第一个参数是DelegateExecution，在表达式环境中默认名称为 execution。 第二个参数传递的是当前流程的名为 myVar 的变量。

要指定执行的 UEL 值表达式， 需要使用 activiti:expression 属性。

```
<serviceTask id="javaService"
         name="My Java Service Task"
         activiti:expression="#{split.ready}" />
```

ready 属性的 getter 方法，getReady（无参数）， 会作用于名为split 的 bean 上。 这个对象会被解析为流程对象和 （如果合适）Spring 环境中的对象。

Implementation 实现

要在流程执行中实现一个调用的类，这个类需要实现 org.activiti.engine.delegate.JavaDelegate 接口， 并在execute 方法中提供对应的业务逻辑。 当流程执行到特定阶段，它会指定方法中定义好的业务逻辑， 并按照默认 BPMN 2.0 中的方式离开节点。 让我们创建一个 java 类的例子，它可以流程变量中字符串转换为大写。 这个类需要实现 org.activiti.engine.delegate.JavaDelegate 接口， 这要求我们实现 execute(DelegateExecution) 方法。 它包含的业务逻辑会被引擎调用。流程实例信息，如流程变量和其他信息， 可以通过 DelegateExecution接口访问和操作（点击对应操作的 javadoc 的链接，获得更多信息）。

```
public class ToUppercase implements JavaDelegate {

  public void execute(DelegateExecution execution) throws Exception {
    String var = (String) execution.getVariable("input");
    var = var.toUpperCase();
    execution.setVariable("input", var);
  }

}
```

注意：serviceTask 定义的 class 只会创建一个 java 类的实例。 所有流程实例都会共享相同的类实例，并调用 execute(DelegateExecution)。 这意味着， 类不能使用任何成员变量，必须是线程安全的，它必须能模拟在不同线程中执行。 这也影响着属性注入的处理方式。

流程定义中引用的类（比如，使用 activiti:class ）不会在部署时实例化。只有当流程第一次执行到使用类的时候， 类的实例才会被创建。如果找不到类，会抛出一个 ActivitiException。这个原因是部署环境（更确切是的 classpath ）和真实环境往往是不同的。 比如当使用 ant 或 业务归档上传到 Activiti Explorer 来发布流程 classpath 没有包含引用的类。

[内部：非公共实现类] 也可以提供实现 org.activiti.engine.impl.pvm.delegate.ActivityBehavior 接口的类。 实现可以访问更强大的 ActivityExecution, 它可以影响流程的流向。注意， 这不是一个很好的实践， 应该尽量避免。所以，建议只有在高级情况下并且你确切知道你要做什么的情况下， 再使用ActivityBehavior 接口。

Field Injection 属性注入

可以为代理类的属性注入数据。支持如下类型的注入：

- 固定的字符串
- 表达式

如果有效的话，数值会通过代理类的setter方法注入，遵循java bean的命名规范（比如 fistName 属性对应 setFirstName(...) 方法）。 如果属性没有对应的 setter 方法，数值会直接注入到私有属性中。 一些环境的 SecurityManager 不允许修改私有属性，所以最好还是把你想注入的属性暴露出对应的 setter 方法来。 无论流程定义中的数据是什么类型，注入目标的属性类型都应该是 org.activiti.engine.delegate.Expression。

下面代码演示了如何把一个常量注入到属性中。 属性注入可以使用 'class' 属性。 注意我们需要定义一个'extensionElements' XML元素， 在声明实际的属性注入之前，这是 BPMN 2.0 XML 格式要求的。

```
<serviceTask id="javaService"
    name="Java service invocation"
    activiti:class="org.activiti.examples.bpmn.servicetask.ToUpperCaseFieldInjected">
    <extensionElements>
      <activiti:field name="text" stringValue="Hello World" />
  </extensionElements>
</serviceTask>
```

ToUpperCaseFieldInjected 类有一个 text 属性， 类型是org.activiti.engine.delegate.Expression。 调 用 text.getValue(execution) 时，会返回定义的字符串 Hello World。

也可以使用长文字（比如，内嵌的 email）， 可以使用 'activiti:string' 子元素：

```
<serviceTask id="javaService"
    name="Java service invocation"
    activiti:class="org.activiti.examples.bpmn.servicetask.ToUpperCaseFieldInjected">
  <extensionElements>
    <activiti:field name="text">
        <activiti:string>
          Hello World
      </activiti:string>
    </activiti:field>
  </extensionElements>
</serviceTask>
```

可以使用表达式，实现在运行期动态解析注入的值。这些表达式可以使用流程变量或 spring 定义的 bean（如果使用了 spring ）。像服务任务实现里说的那样，服务任务中的java类实例会在所有流程实例中共享。 为了动态注入属性的值，我们可以在org.activiti.engine.delegate.Expression 中使用值和方法表达式，它会使用传递给 execute 方法的 DelegateExecution 参数进行解析。

```
<serviceTask id="javaService" name="Java service invocation"
  activiti:class="org.activiti.examples.bpmn.servicetask.ReverseStringsFieldInjected">

  <extensionElements>
    <activiti:field name="text1">
      <activiti:expression>${genderBean.getGenderString(gender)}</activiti:expression>
    </activiti:field>
    <activiti:field name="text2">
       <activiti:expression>Hello ${gender == 'male' ? 'Mr.' : 'Mrs.'} ${name}</activiti:expression>
    </activiti:field>
  </ extensionElements>
</ serviceTask>
```

下面的例子中，注入了表达式，并使用在传入的当前DelegateExecution解析它们。 完整代码可以参考

org.activiti.examples.bpmn.servicetask.JavaServiceTaskTest.testExpressionFieldInjection。

```
public class ReverseStringsFieldInjected implements JavaDelegate {

  private Expression text1;
  private Expression text2;

  public void execute(DelegateExecution execution) {
    String value1 = (String) text1.getValue(execution);
    execution.setVariable("var1", new StringBuffer(value1).reverse().toString());

    String value2 = (String) text2.getValue(execution);
    execution.setVariable("var2", new StringBuffer(value2).reverse().toString());
  }
}
```

另外，你也可以把表达式设置成一个属性，而不是字元素，让XML更简单一些。

```
<activiti:field name="text1" expression="${genderBean.getGenderString(gender)}" />
<activiti:field name="text1" expression="Hello ${gender == 'male' ? 'Mr.' : 'Mrs.'} ${name}" />
```

因为 java 类实例会被重用，注入只会发生一次，当服务任务调用第一次的时候。 当你的代码中的属性改变了， 值也不会重新注入，所以你应该把它们看做是不变的，不用修改它们。

Service task results 服务任务结果

服务流程返回的结果（使用表达式的服务任务）可以分配给已经存在的或新的流程变量， 可以通过指定服务任务定义的 'activiti:resultVariable' 属性来实现。 指定的路程比那两的值会被服务流程的返回结果覆盖。如果没有指定返回变量名，就会忽略返回结果。

```
<serviceTask id="aMethodExpressionServiceTask"
    activiti:expression="#{myService.doSomething()}"
    activiti:resultVariable="myVar" />
```

在上面的例子中，服务流程的返回值（在'myService'上调用 'doSomething()' 方法的返回值， myService 可能是流程变量，也可能是 spring 的 bean ），会设置到名为 'myVar' 的流程变量里，在服务执行完成之后。

Handling exceptions处理异常

执行自定义逻辑时，常常需要捕获对应的业务异常，在流程内部进行处理。 activiti 提供了不同的方式来处理这个问题

Throwing BPMN Errors 抛出BPMN Errors

可以在服务任务或脚本任务的代码里抛出 BPMN error。 为了实现这个，要从 JavaDelegate，脚本，表达式和代理表达式中抛出名为 BpmnError的特殊 ActivitiExeption。 引擎会捕获这个异常，把它转发到对应的错误处理中。比如，边界错误事件或错误事件子流程。

```
public class ThrowBpmnErrorDelegate implements JavaDelegate {

  public void execute(DelegateExecution execution) throws Exception {
    try {
      executeBusinessLogic();
    } catch (BusinessException e) {
      throw new BpmnError("BusinessExceptionOccurred");
    }
  }

}
```

构造参数是错误代码，会被用来决定 哪个错误处理器会来响应这个错误。 参考边界错误事件 获得更多捕获 BPMN error 的信息。

这个机制应该只用于业务失败， 它应该被流程定义中设置的边界错误事件或错误事件子流程处理。技术上的错误应该使用其他异常类型，通常不会在流程里处理。

Exception Sequence Flow 异常顺序流

[内部，公开实现类] 另一种选择是在一些异常发生时，让路程进入其他路径。下面的代码演示了如何实现。

```
<serviceTask id="javaService"
  name="Java service invocation"
  activiti:class="org.activiti.ThrowsExceptionBehavior">
</serviceTask>

<sequenceFlow id="no-exception" sourceRef="javaService" targetRef="theEnd" />
<sequenceFlow id="exception" sourceRef="javaService" targetRef="fixException" />
```

这里的服务任务有两个外出顺序流，分别叫 exception 和 no-exception 。异常出现时会使用顺序流的 id 来决定流向

```
public class ThrowsExceptionBehavior implements ActivityBehavior {

  public void execute(ActivityExecution execution) throws Exception {
    String var = (String) execution.getVariable("var");

    PvmTransition transition = null;
    try {
      executeLogic(var);
      transition = execution.getActivity().findOutgoingTransition("no-exception");
    } catch (Exception e) {
      transition = execution.getActivity().findOutgoingTransition("exception");
    }
    execution.take(transition);
  }

}
```

Using an Activiti service from within a JavaDelegate 在 JavaDelegate 里使用 activiti 服务

一些场景下，需要在 java 服务任务中使用 activiti 服务 （比如，通过 RuntimeService 启动流 程实例，而 callActivity 不满足你的需求）。 org.activiti.engine.delegate.DelegateExecution 允许通过 org.activiti.engine.EngineServices 接口直接获得这些服务：

```
public class StartProcessInstanceTestDelegate implements JavaDelegate {

  public void execute(DelegateExecution execution) throws Exception {
    RuntimeService runtimeService = execution.getEngineServices().getRuntimeService();
    runtimeService.startProcessInstanceByKey("myProcess");
  }

}
```

所有 activiti 服务的 API 都可以通过这个接口获得。

使用这些 API 调用出现的所有数据改变，都是在当前事务中的。在像spring 和 CDI 这样的依赖注入环境也会起作用，无论是否启用了 JTA 数据源。 比如，下面的代码功能与上面的代码一致， 这是 RuntimeService 是通过依赖注入获得的，而不是通过 org.activiti.engine.EngineServices 接口。

```
@Component("startProcessInstanceDelegate")
public class StartProcessInstanceTestDelegateWithInjection {

    @Autowired
    private RuntimeService runtimeService;

    public void startProcess() {
      runtimeService.startProcessInstanceByKey("oneTaskProcess");
    }

}
```

重要技术提示：因为服务调用是在当前事务里， 数据的产生或改变，在服务任务执行完之前，还没有提交到数据库。 所有 API 对于数据库数据的操作，意味着未提交的操作在服务任务的 API 调用中都是不可见的

Web Service Task

[试验]

Description 描述

Web Service任务可以用来同步调用一个外部的Web service。

Graphical notation 图形标记

Web Service 任务与 Java服务任务显示效果一样。



XML representation 内容

要使用 Web Service 我们需要导入它的操作和类型。 可以自动使用import 标签来指定 Web Service 的 WSDL：

```
<import importType="http://schemas.xmlsoap.org/wsdl/"
    location="http://localhost:63081/counter?wsdl"
    namespace="http://webservice.activiti.org/" />
```

上面的声明告诉 activiti 导入 WSDL 定义，但没有创建 item 定义和消息。 假设我们想调用一个名为 'prettyPrint' 的方法， 我们必须创建为请求和响应信息对应的消息和 item 定义：

```
<message id="prettyPrintCountRequestMessage" itemRef="tns:prettyPrintCountRequestItem" />
<message id="prettyPrintCountResponseMessage" itemRef="tns:prettyPrintCountResponseItem" />

<itemDefinition id="prettyPrintCountRequestItem" structureRef="counter:prettyPrintCount" />
<itemDefinition id="prettyPrintCountResponseItem" structureRef="counter:prettyPrintCountResponse" />
```

在申请服务任务之前，我们必须定义实际引用 Web Service 的 BPMN 接口和操作。 基本上，我们定义接口和必要的操作。 对每个奥做我们都会重用上面定义的信息作为输入和输出。 比如，下面定义了 'counter' 接口和 'prettyPrintCountOperation' 操作：

```
<interface name="Counter Interface" implementationRef="counter:Counter">
        <operation id="prettyPrintCountOperation" name="prettyPrintCount Operation"
                        implementationRef="counter:prettyPrintCount">
                <inMessageRef>tns:prettyPrintCountRequestMessage</inMessageRef>
                <outMessageRef>tns:prettyPrintCountResponseMessage</outMessageRef>
        </operation>
</interface>
```

然后我们可以定义 Web Service 任务使用 ##WebService 实现， 并引用 Web Service 操作。

```
<serviceTask id="webService"
    name="Web service invocation"
    implementation="##WebService"
    operationRef="tns:prettyPrintCountOperation">
```

Web Service Task IO Specification 规范

除非我们使用简化方式处理数据输入和输出关联（如下所示）， 每个 Web Service 任务可以定义任务的输入输出 IO 规范。配置方式与 BPMN 2.0 完全兼容，下面格式化后的例子，我们根据之前定义 item 定义，定义了输入和输出。

```
<ioSpecification>
        <dataInput itemSubjectRef="tns:prettyPrintCountRequestItem" id="dataInputOfServiceTask" />
        <dataOutput itemSubjectRef="tns:prettyPrintCountResponseItem" id="dataOutputOfServiceTask" />
        <inputSet>
                <dataInputRefs>dataInputOfServiceTask</dataInputRefs>
        </inputSet>
        <outputSet>
                <dataOutputRefs>dataOutputOfServiceTask</dataOutputRefs>
        </outputSet>
</ioSpecification>
```

Web Service Task data input associations 任务数据输入关联

有两种方式指定数据输入关联：

- 使用表达式
- 使用简化方式

要使用表达式指定数据输入关联，我们需要定义来源和目的 item，并指定每个 item 属性之间的对应关系。 下面的例子中我们分配了这些 item 的前缀和后缀：

```
<dataInputAssociation>
        <sourceRef>dataInputOfProcess</sourceRef>
        <targetRef>dataInputOfServiceTask</targetRef>
        <assignment>
                <from>${dataInputOfProcess.prefix}</from>
                <to>${dataInputOfServiceTask.prefix}</to>
        </assignment>
        <assignment>
                <from>${dataInputOfProcess.suffix}</from>
                <to>${dataInputOfServiceTask.suffix}</to>
        </assignment>
</dataInputAssociation>
```

另外，我们可以使用更简单的简化方式。'sourceRef' 元素是 activiti 的变量名，'targetRef' 元素是 item 定义的一个属性。在下面的例子中，我们把 'PrefixVariable' 变量的值分配给 'field' 属性， 把 'SuffixVariable' 变量的值分配给 'suffix' 属性。

```
<dataInputAssociation>
        <sourceRef>PrefixVariable</sourceRef>
        <targetRef>prefix</targetRef>
</dataInputAssociation>
<dataInputAssociation>
        <sourceRef>SuffixVariable</sourceRef>
        <targetRef>suffix</targetRef>
</dataInputAssociation>
```

Web Service Task data output associations 任务数据输出关联

有两种方式指定数据输出关联：

- 使用表达式
- 使用简化方式

要使用表达式指定数据输出关联，我们需要定义目的变量和来源表达式。 方法和数据输入关联完全一样：

```
<dataOutputAssociation>
        <targetRef>dataOutputOfProcess</targetRef>
        <transformation>${dataOutputOfServiceTask.prettyPrint}</transformation>
</dataOutputAssociation>
```

另外，我们可以使用更简单的简化方式。'sourceRef' 元素是 item 定义的一个属性，'targetRef' 元素是 activiti 的变量名。 方法和数据输入关联完全一样：

```
<dataOutputAssociation>
        <sourceRef>prettyPrint</sourceRef>
        <targetRef>OutputVariable</targetRef>
</dataOutputAssociation>
```

Business Rule Task 业务规则任务

[试验]

Description 描述

业务规则用户用来同步执行一个或多个规则。activiti 使用 drools 规则引擎执行业务规则。目前，包含业务规则的 .drl 文件必须和流程定义一起发布，流程定义里包含了执行这些规则的业务规则任务。意味着流程使用的所有.drl文件都必须打包在流程BAR文件里，比如任务表单。更多使用Drools Expert 创建业务规则的信息，请参考 JBoss Drools 的文档。如果想要使用你的规则任务的实现，比如，因为你想用不同方式使用 drools，或你想使用完全不同的规则引擎，你可以使用 BusinessRuleTask 上的 class 或表达式属性，它用起来就和 ServiceTask 一样。

Graphical notation 图形标记

业务规则任务使用一个表格小图标进行显示。



XML representation 内容

要执行部署流程定义的 BAR 文件中的一个或多个业务规则，我们需要定义输入和输出变量。对于输入变量定义，可以使用逗号分隔的一些流程变量。 输出变量定义智能包含一个变量名，它会把执行业务规则后返回的对象保存到对应的流程变量中。 注意，结果变量会包含一个对象列表。如果没有指定输出变量名称，默认会使用 org.activiti.engine.rules.OUTPUT。

下面的业务规则任务会执行和流程定义一起部署的素有业务规则：

```
<process id="simpleBusinessRuleProcess">

  <startEvent id="theStart" />
  <sequenceFlow sourceRef="theStart" targetRef="businessRuleTask" />

  <businessRuleTask id="businessRuleTask" activiti:ruleVariablesInput="${order}"
      activiti:resultVariable="rulesOutput" />

  <sequenceFlow sourceRef="businessRuleTask" targetRef="theEnd" />

  <endEvent id="theEnd" />

</process>
```

业务规则任务也可以配置成只执行部署的 .drl 文件中的一些规则。 这时要设置逗号分隔的规则名。

```
<businessRuleTask id="businessRuleTask" activiti:ruleVariablesInput="${order}"
  activiti:rules="rule1, rule2" />
```

这时，只会执行 rule1 和 rule2。

你也可以定义哪些规则不用执行。

```
<businessRuleTask id="businessRuleTask" activiti:ruleVariablesInput="${order}"
  activiti:rules="rule1, rule2" exclude="true" />
```

这时除了 rule1 和 rule2 以外，所有部署到流程定义同一个BAR文件中的规则都会执行。

像之前提到的，可以用一个选项修改 BusinessRuleTask 的实现：

注意 BusinessRuleTask 的功能和 ServiceTask 一样，但是我们使用BusinessRuleTask 的图标来表示 我们在这里要执行业务规则。

Email Task

activiti 强化了业务流程，支持了自动邮件任务，它可以发送邮件给一个或多个参与者， 包括支持 cc, bcc, HTML 内容等等。 注意邮件任务不是 BPMN 2.0 规范定义的官方任务。 （它也没有对应的图标）。 因此，activiti 中邮件任务是用专门的服务任务实现的。

Mail server configuration 邮件服务器配置

activiti 引擎要通过支持SMTP功能的外部邮件服务器发送邮件。 为了实际发送邮件，引擎需要知道如何访问邮件服务器。下面的配置可以设置到activiti.cfg.xml 配置文件中：

Table 8.1. Mail server configuration

属性	是否需要?	描述
mailServerHost	no	mail服务器名称 (如 mail.mycorp.com). 默认是 localhost

mailServerPort	yes, 但不是在默认端口	邮件服务器上的SMTP传输端口。默认为25
mailServerDefaultFrom	no	如果用户没有指定发送邮件的邮件地址，默认设置的发送者的邮件地址。默认为activiti@activiti.org
mailServerUsername	如果服务器需要	一些邮件服务器需要认证才能发送邮件。默认不设置。
mailServerPassword	如果服务器需要	一些邮件服务器需要认证才能发送邮件。默认不设置。
mailServerUseSSL	如果服务器需要	一些邮件服务器需要ssl交互。默认为false。
mailServerUseTLS	如果服务器需要	一些邮件服务器（比如gmail）需要支持TLS。默认为false。

Defining an Email Task 定义

邮件任务是一个专用的服务任务， 这个服务任务的type设置为 'mail'。

```
<serviceTask id="sendMail" activiti:type="mail">
```

邮件任务是通过属性注入进行配置的。 所有这些属性都可以使用 EL 表达式，可以在流程执行中解析。 下面的属性都可以设置：

Table 8.2. Mail task configuration

属性	是否需要?	描述
to	yes	邮件的接受者。可以使用逗号分 隔多个接受者
from	no	邮件发送者的地址。如果不提 供，会使用默认配置的地址
subject	no	邮件的主题
cc	no	邮件抄送人。可以使用逗号分隔 多个接收者
bcc	no	邮件暗送人。可以使用逗号分隔 多个接收者
charset	no	可以修改邮件的字符集，对很多非英语语言是必须设置的。
html	no	作为邮件内容的HTML
text	no	邮件的内容，在需要使用原始文 字（非富文本）的邮件时使用。可以与html一起使用，对于不支持富客户端的邮件客户端。 客户端会降级到仅显示文本的方式。
htmlVar	no	使用对应的流程变量作为e-mail的内容。它和html的不同之处是它内容中包含的表达式会在 mail 任务发送之前被替换掉。
textVar	no	使用对应的流程变量作为e-mail 的纯文本内容。它和 html 的不同之处是它内容中包含的表达式会 在mail任务发送之前被替换掉
ignoreException	no	处理邮件失败时，是否忽略异常，不抛出ActivitiException，默认为false。
exceptionVariableName	no	当设置了ignoreException =true 处理 email 时不抛出异常，可以指定一个变量名来存储失败信息。

Example usage 使用实例

下面的XML演示了使用邮件任务的例子

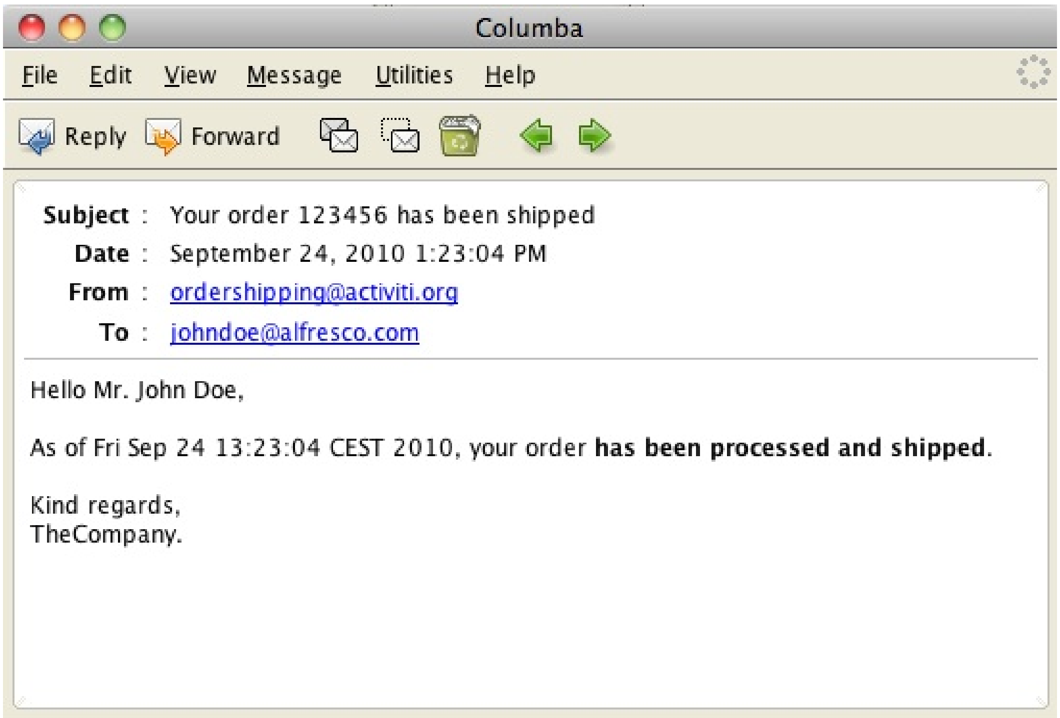
```
<serviceTask id="sendMail" activiti:type="mail">
  <extensionElements>
    <activiti:field name="from" stringValue="order-shipping@thecompany.com" />
    <activiti:field name="to" expression="${recipient}" />
    <activiti:field name="subject" expression="Your order ${orderId} has been shipped" />
    <activiti:field name="html">
      <activiti:expression>
        <![CDATA[
          <html>
            <body>
              Hello ${male ? 'Mr.' : 'Mrs.' } ${recipientName},<br/><br/>

              As of ${now}, your order has been <b>processed and shipped</b>.<br/><br/>

              Kind regards,<br/>

              TheCompany.
            </body>
          </html>
        ]]>
      </activiti:expression>
    </activiti:field>
  </extensionElements>
</serviceTask>
```

结果如下：



Mule Task

mule 任务可以向 mule 发送消息，以强化activiti的集成能力。注意mule 任务不是 BPMN 2.0 规范定义的官方任务。它也没有对应的图标）。 因此，activiti 中 mule 任务是用专门的服务任务实现的。

Defining an Mule Task 定义

mule 任务是一个专用的服务任务， 这个服务任务的 type 设置为 'mule'。

```
<serviceTask id="sendMule" activiti:type="mule">
```

mule 任务是通过属性注入进行配置的。 所有这些属性都可以使用 EL 表达式，可以在流程执行中解析。 下面的属性都可以设置：

Table 8.3. Mule server configuration

属性	是否需要?	描述
endpointUrl	yes	希望调用的Mule终端
language	yes	你要使用解析荷载表达式 （payloadExpression）属性的语言
payloadExpression	yes	作为消息荷载的表达式。
resultVariable	no	将要保存调用结果的变量名称

Example usage 应用实例

下面是一个使用 mule 任务的例子

```
<extensionElements>
<activiti:field name="endpointUrl">
  <activiti:string>vm://in</activiti:string>
</activiti:field>
<activiti:field name="language">
  <activiti:string>juel</activiti:string>
</activiti:field>
<activiti:field name="payloadExpression">
  <activiti:string>"hi"</activiti:string>
</activiti:field>
<activiti:field name="resultVariable">
  <activiti:string>theVariable</activiti:string>
</activiti:field>
</extensionElements>
```

Camel Task

Camel 任务可以从 Camel 发送和介绍消息，由此强化了 activiti 的集成功能。 注意 camel 任务不是 BPMN 2.0 规范定义的官方任务。 （它也没有对应的图标）。 在 activiti 中，camel 任务时由专用的服务任务实现的。 要使用 camel 任务功能时，也要记得吧 activiti camel 包含到项目里。

Defining a Camel Task 定义

camel 任务是一个专用的服务任务，这个服务任务的type设置为 'camel'。

```
<serviceTask id="sendCamel" activiti:type="camel">
```

流程定义只需要在服务任务中定义 camel 类型。 集成逻辑都会代理给camel 容器。默认 activiti 引擎会在 spring 容器中查找 camelContext bean。 camelContext 定义了 camel 容器 加载的路由规则。下面的例子中路由规则是从指定的java包下加载的。 但是你也可以通过 spring 配置直接定义路由规则。

```
<camelContext id="camelContext" xmlns="http://camel.apache.org/schema/spring">
  <packageScan>
    <package>org.activiti.camel.route</package>
  </packageScan>
</camelContext>
```

如果想了解更多关于 camel 路由的信息，可以访问 Camel 的网站在这里只通过很小的例子演示了基础的概念。 在第一个例

子中，我们会通过 activiti 工作流实现最简单的Camel调用。我们称其为 SimpleCamelCall。

如果想定义多个 Camel 环境 bean，并且（或者）想使用不同的 bean 名称，可以重载 CamelTask 的定义，如下所示：

```
<serviceTask id="serviceTask1" activiti:type="camel">
  <extensionElements>
    <activiti:field name="camelContext" stringValue="customCamelContext" />
  </extensionElements>
</serviceTask>
```

Simple Camel Call example 简单Camel调用

这个例子对应的文件都可以在 activiti camel 模块的 org.activiti.camel.examples.simpleCamelCall 包下找到。我们的目标是简单激活一个特定的 camel 路由。 首先，我们需要一个 Spring 环境，它要包含之前介绍的路由。这些文件的目的如下：

```
<camelContext id="camelContext" xmlns="http://camel.apache.org/schema/spring">
  <packageScan>
    <package>org.activiti.camel.examples.simpleCamelCall</package>
  </packageScan>
</camelContext>
```

包含名为 SimpleCamelCallRoute 的路由的类文件，放在 PackageScan标签的扫描目录下。 下面就是路由的定义：

```
public class SimpleCamelCallRoute extends RouteBuilder {

  @Override
  public void configure() throws Exception {
    from("activiti:SimpleCamelCallProcess:simpleCall").to("log:org.activiti.camel.examples.SimpleCamelCall");
  }
}
```

这个规则仅仅打印消息体，不会做其他事情。注意终端的格式。它包含三部分：

Table 8.4. Endpoint URL parts:

Endpoint Url 部分	描述
activiti	引用 Activiti endpoint
SimpleCamelCallProcess	进程名字
simpleCall	流程中的 Camel 服务

OK，我们的规则已经配置好，也可以让 Camel 使用了。 现在看工作流部分。工作流看起来像这样：

```
<process id="SimpleCamelCallProcess">
  <startEvent id="start"/>
  <sequenceFlow id="flow1" sourceRef="start" targetRef="simpleCall"/>

  <serviceTask id="simpleCall" activiti:type="camel"/>

  <sequenceFlow id="flow2" sourceRef="simpleCall" targetRef="end"/>
  <endEvent id="end"/>
</process>
```

在 serviceTask 部分，它只注明服务的类型是 Camel，目标规则名为simpleCall。这与上面的 activiti 终端相匹配。初始化流程后，我们会看到一个空的日志。 好，我们已经完成了这个最简单的例子了

Ping Pong example 乒乓实例

我们的例子成功执行了，但是 Camel 和 Activiti 之间没有任何交互，而且这样做也没有任何优势。在这个例子里，我们尝试向 Camel 发送和接收数据。 我们发送一个字符串，camel 进行一些处理，然后返回结果。 发送部分很简单，我们把变量里的消息发送给 camel。这里是我们的调用代码

```
@Deployment
public void testPingPong() {
  Map<String, Object> variables = new HashMap<String, Object>();

  variables.put("input", "Hello");
  Map<String, String> outputMap = new HashMap<String, String>();
  variables.put("outputMap", outputMap);

  runtimeService.startProcessInstanceByKey("PingPongProcess", variables);
  assertEquals(1, outputMap.size());
  assertNotNull(outputMap.get("outputValue"));
  assertEquals("Hello World", outputMap.get("outputValue"));
}
```

变量"input"是 Camel 规则的实际输入，outputMap 会记录 camel 返回的结果。流程应该像是这样：

```
<process id="PingPongProcess">
  <startEvent id="start"/>
  <sequenceFlow id="flow1" sourceRef="start" targetRef="ping"/>
  <serviceTask id="ping" activiti:type="camel"/>
  <sequenceFlow id="flow2" sourceRef="ping" targetRef="saveOutput"/>
  <serviceTask id="saveOutput"  activiti:class="org.activiti.camel.examples.pingPong.SaveOutput" />
  <sequenceFlow id="flow3" sourceRef="saveOutput" targetRef="end"/>
  <endEvent id="end"/>
</process>
```

注意，SaveOuput 这个 serviceTask，会把"Output"变量的值从上下文保存到上面提到的 OutputMap 中。 现在，我们必须了解变量是如何发送给 Camel，再返回的。这里就要涉及到 camel 实际执行的行为了。 变量提交给 camel 的方法是由 CamelBehavior 控制的。这里我们使用默认的配置，其他的会在后面提及。 使用这些代码，我们就可以配置一个期望的 camel 行为：

```
<serviceTask id="serviceTask1" activiti:type="camel">
  <extensionElements>
    <activiti:field name="camelBehaviorClass" stringValue="org.activiti.camel.impl.CamelBehaviorCamelBodyImpl" />
  </extensionElements>
</serviceTask>
```

如果你没有特别指定一个行为，就会使用 org.activiti.camel.impl.CamelBehaviorDefaultImpl。 这个行为会把变量复制成名称相同的 Camel 属性。 在返回时，无论选择什么行为，如果 camel 消息体是一个 map，每个元素都会复制成一个变量， 否则整个对象会复制到指定名称为"camelBody"的变量中。 了解这些后， 就可以看看我们第二个例子的 camel 规则了：

```
@Override
public void configure() throws Exception {
  from("activiti:PingPongProcess:ping").transform().simple("${property.input} World");
}
```

在这个规则中，字符串"world"会被添加到"input"属性的后面，结果会写入消息体。 这时可以检查 javaServiceTask 中的"camelBody"变量，复制到"outputMap"中，并在 testcase 进行判断。现在这个例子是在默认的行为下运行的，然后我们看一起其他的方案。在启动的所有 camel 规则中，流程实例 id 会复制到 camel 的名为"PROCESS_ID_PROPERTY"的属性中。 后续可以用它关联流程实例和 camel 规则。他也可以在 camel规则中直接使用。

Activiti 中可以使用三种不同的行为。这些行为可以通过在规则URL中指定对应的环节来实现覆盖。 这里有一个在URL中覆盖现存行为的例子：

```
from("activiti:asyncCamelProcess:serviceTaskAsync2?copyVariablesToProperties=true").
```

下面的表格提供了三种 camel 行为的概述：

Table 8.5. Existing camel behaviours:

行为	**Url**	描述
CamelBehaviorDefaultImpl	copyVariablesToProperties	复制 Activiti 属性作为 Camel 属性
CamelBehaviorCamelBodyImpl	copyCamelBodyToBody	只把名为 "camelBody" Activiti变量复制成camel的消息体
CamelBehaviorBodyAsMapImpl	copyVariablesToBodyAsMap	把activiti的所有变量复制到一个map里，作为Camel的消息体

上面的表格解释和 activiti 变量如何传递给 camel。下面的表格解释和camel 的变量如何返回给 activiti。 它只能配置在规则URL中。

Table 8.6. Existing camel behaviours:

Url	描述	
Default	如果Camel消息体是一个map，把 每个元素复制成activiti的变量，否则把整个camel消息体作 为activiti的"camelBody"变量。	
copyVariablesFromProperties	将Camel属性以相同名称复制为Activiti变量	
copyCamelBodyToBodyAsString	和默认一样，但是如果camel消息体不是map时，先把它转换成字符串，再设置为 "camelBody"。	
copyVariablesFromHeader	额外把camel头部以相同名称复制成Activiti变量	

例子的源码放在 activiti-camel 模块的org.activiti.camel.examples.pingPong 包下。

Asynchronous Ping Pong example 异步乒乓实例

之前的例子都是同步的。流程会等到 camel 规则返回之后才会停止。 一些情况下，我们需要 activiti 工作流继续运行。这时 camelServiceTask 的异步功能就特别有用。 你可以通过设置 camelServiceTask 的 async 属性来启用这个功能。

```
<serviceTask id="serviceAsyncPing" activiti:type="camel" activiti:async="true"/>
```

通过设置这个功能，camel 规则会被 activiti 的 jobExecutor 异步执行。 当你在 camel 规则中定义了一个队列，activiti 流程会在camelServiceTask 执行时继续运行。 camel 规则会以完全异步的方式执行。 如果你想在什么地方等待 camelServiceTask 的返回值，你可以使用一个 receiveTask。

```
<receiveTask id="receiveAsyncPing" name="Wait State" />
```

流程实例会等到接收一个 signal，比如来自 camel。在 camel 中你可以发送一个 signal 给流程实例，通过对应的 activiti 终端发送消息

```
from("activiti:asyncPingProcess:serviceAsyncPing").to("activiti:asyncPingProcess:receiveAsyncPing");
```

对于一个常用的终端，会使用冒号分隔的三个部分：

- 常量字符串"activiti"
- 流程名称
- 接收任务名

Instantiate workflow from Camel route 从camel规则中实例化工作流

之前的所有例子中，activiti 工作流会先启动，然后在流程中启动 camel 规则。 也可以使用另外一种方法。在已经启动的 camel 规则中启动一个工作流。 这会触发一个 receiveTask 十分类似，除了最后的部分。这是一个实例规则：

```
from("direct:start").to("activiti:camelProcess");
```

我们看到 url 有两个部分，第一个部分是常量字符串"activiti"，第二部分是流程的名称。很明显，流程应该已经部署完成，并且是可以启动的。

也可以设置流程发起人到 Camel 头提供的身份验证的用户 ID 。为了实现这个，发起人变量必须在流程定义中指定的：

```
<startEvent id="start" activiti:initiator="initiator" />
```

接着 用户 Id 包含在 Camel 头 名字叫 CamelProcessInitiatorHeader， 定义如下

```
from("direct:startWithInitiatorHeader")
    .setHeader("CamelProcessInitiatorHeader", constant("kermit"))
    .to("activiti:InitiatorCamelCallProcess?processInitiatorHeaderName=CamelProcessInitiatorHeader");
```

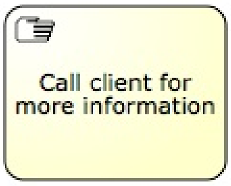
Manual Task 手工任务

Description 描述

手工任务定义了 BPM 引擎外部的任务。 用来表示工作需要某人完成，而引擎不需要知道，也没有对应的系统和 UI 接口。对于引擎，手工任务是直接通过的活动， 流程到达它之后会自动向下执行。

Graphical notation 图形标记

手工任务显示为一个圆角矩形，左上角是一个手型小图标。



XML representation 内容

```
<manualTask id="myManualTask" name="Call client for more information" />
```

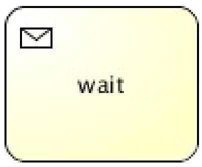
Java Receive Task

Description 描述

接收任务是一个简单任务，它会等待对应消息的到达。当前，我们只实现了这个任务的 java 语义。 当流程达到接收任务，流程状态会保存到存储里。 意味着流程会等待在这个等待状态， 直到引擎接收了一个特定的消息， 这会触发流程穿过接收任务继续执行。

Graphical notation 图形标记

接收任务显示为一个任务（圆角矩形），右上角有一个消息小标记。消息是白色的（黑色图标表示发送语义）



XML representation 内容

```
<receiveTask id="waitState" name="wait" />
```

要在接收任务等待的流程实例继续执行， 可以调用runtimeService.signal(executionId)， 传递接收任务上流程的 id。 下面的代码演示了实际是如何工作的：

```
ProcessInstance pi = runtimeService.startProcessInstanceByKey("receiveTask");
Execution execution = runtimeService.createExecutionQuery()
  .processInstanceId(pi.getId())
  .activityId("waitState")
  .singleResult();
assertNotNull(execution);

runtimeService.signal(execution.getId());
```

Shell Task

Description 描述

shell任务可以执行shell脚本和命令。 注意 shell 任务不是 BPMN 2.0 规范定义的官方任务。（它也没有对应的图标）。

defining a shell task 定义

shell任务是一个专用的服务任务， 这个服务任务的type设置为'shell'

```
<serviceTask id="shellEcho" activiti:type="shell">
```

shell 任务使用属性注入进行配置。 所有属性都可以包含 EL 表达式，会在流程执行过程中解析。可以配置以下属性：

Table 8.7. Shell task parameter configuration

属性	是否需要?	Type	描述	默认
command	yes	String	执行的shell命令	
arg0-5	no	String	参数0至5	
wait	no	true/false	是否需要等待到shell进程结束	true
redirectError	no	true/false	标准错误与标准输出合并	false
cleanEnv	no	true/false	shell进行不继承当前环境	false
outputVariable	no	String	包含输出变量名称	输出不记录
errorCodeVariable	no	String	包含结果错误代码的变量名	不会注册错误级别
directory	no	String	shell进程的默认目录	当前目录

Example usage 应用实例

下面的代码演示了使用 shell 任务的实例。它会执行 shell 脚本"cmd /c echo EchoTest"，等到它结束，再把输出结果保存到 resultVar 中。

```
<serviceTask id="shellEcho" activiti:type="shell" >
  <extensionElements>
    <activiti:field name="command" stringValue="cmd" />
    <activiti:field name="arg1" stringValue="/c" />
    <activiti:field name="arg2" stringValue="echo" />
    <activiti:field name="arg3" stringValue="EchoTest" />
    <activiti:field name="wait" stringValue="true" />
    <activiti:field name="outputVariable" stringValue="resultVar" />
  </extensionElements>
</serviceTask>
```

Execution listener 执行监听器

兼容性提醒：在发布 5.3 后，我们发现执行监听器， 任务监听器，表达式还是非公开 API。这些类在 org.activiti.engine.impl... 的子包， 包名中有一个 impl。 org.activiti.engine.impl.pvm.delegate.ExecutionListener, org.activiti.engine.impl.pvm.delegate.TaskListener 和 org.activiti.engine.impl.pvm.el.Expression 已经废弃了。 从现在开始，应该使用 org.activiti.engine.delegate.ExecutionListener, org.activiti.engine.delegate.TaskListener 和 org.activiti.engine.delegate.Expression。 在新的公开 API 中， 删除了 ExecutionListenerExecution.getEventSource()。 因为已经设置了废弃编译警告，所以已存的代码应该可以正常运行。但是要考虑切换到新的公共API接口 （包名中没有.impl.）。

执行监听器可以执行外部 java 代码或执行表达式，当流程定义中发生了某个事件。 可以捕获的事件有：

- 流程实例的启动和结束。
- 选中一条连线。
- 节点的开始和结束。
- 网关的开始和结束。
- 中间事件的开始和结束。
- 开始时间结束或结束事件开始。

下面的流程定义包含了3个流程监听器：

```
<process id="executionListenersProcess">

   <extensionElements>
     <activiti:executionListener class="org.activiti.examples.bpmn.executionlistener.ExampleExecutionListenerOne" ever
   </extensionElements>

   <startEvent id="theStart" />
```

```
    <sequenceFlow sourceRef="theStart" targetRef="firstTask" />

    <userTask id="firstTask" />
    <sequenceFlow sourceRef="firstTask" targetRef="secondTask">
    <extensionElements>
      <activiti:executionListener class="org.activiti.examples.bpmn.executionListener.ExampleExecutionListenerTwo" />
    </extensionElements>
    </sequenceFlow>

    <userTask id="secondTask" >
    <extensionElements>
      <activiti:executionListener expression="${myPojo.myMethod(execution.event)}" event="end" />
    </extensionElements>
    </userTask>
    <sequenceFlow sourceRef="secondTask" targetRef="thirdTask" />

    <userTask id="thirdTask" />
    <sequenceFlow sourceRef="thirdTask" targetRef="theEnd" />

    <endEvent id="theEnd" />

  </process>
```

第一个流程监听器监听流程开始。监听器是一个外部 java 类（像 是 ExampleExecutionListenerOne）， 需要实现 org.activiti.engine.delegate.ExecutionListener 接口。 当事件发生时（这里是 end 事件）， 会调用 notify(ExecutionListenerExecution execution) 方法。

```
public class ExampleExecutionListenerOne implements ExecutionListener {

  public void notify(ExecutionListenerExecution execution) throws Exception {
    execution.setVariable("variableSetInExecutionListener", "firstValue");
    execution.setVariable("eventReceived", execution.getEventName());
  }
}
```

也可以使用实现 org.activiti.engine.delegate.JavaDelegate 接口的代理类。 代理类可以在结构中重用，比如 serviceTask 的代理。

第二个流程监听器在连线执行时调用。注意这个 listener 元素不能定义event， 因为连线只能触发 take 事件。 为连线定义的监听器的event属性会被忽略。

最后一个流程监听器在节点 secondTask 结束时调用。这里使用expression 代替 class 来在事件触发时执行/调用。

```
<activiti:executionListener expression="${myPojo.myMethod(execution.eventName)}" event="end" />
```

和其他表达式一样，流程变量可以处理和使用。因为流程实现对象有一个保存事件名称的属性， 可以在方法中使用 execution.eventName 获的事件名称。

流程监听器也支持使用 delegateExpression, 和服务任务相同。

```
<activiti:executionListener event="start" delegateExpression="${myExecutionListenerBean}" />
```

在 activiti 5.12 中，我们也介绍了新的流程监听器， org.activiti.engine.impl.bpmn.listener.ScriptExecutionListener。 这个脚本流程监听器可以为某个流程监听事件执行一段脚本。

```
<activiti:executionListener event="start" class="org.activiti.engine.impl.bpmn.listener.ScriptExecutionListener" >
  <activiti:field name="script">
    <activiti:string>
```

```
      def bar = "BAR";  // local variable
      foo = "FOO"; // pushes variable to execution context
      execution.setVariable("var1", "test"); // test access to execution instance
      bar // implicit return value
    </activiti:string>
  </activiti:field>
  <activiti:field name="language" stringValue="groovy" />
  <activiti:field name="resultVariable" stringValue="myVar" />
<activiti:executionListener>
```

Field injection on execution listeners 流程监听器的属性注入

使用流程监听器时，可以配置class属性，可以使用属性注入。这和使用服务任务属性注入相同， 参考它可以获得属性注入的很多信息。

下面的代码演示了使用了属性注入的流程监听器的流程的简单例子。

```
 <process id="executionListenersProcess">
    <extensionElements>
      <activiti:executionListener class="org.activiti.examples.bpmn.executionListener.ExampleFieldInjectedExecutionList
        <activiti:field name="fixedValue" stringValue="Yes, I am " />
        <activiti:field name="dynamicValue" expression="${myVar}" />
      </activiti:executionListener>
    </extensionElements>

    <startEvent id="theStart" />
    <sequenceFlow sourceRef="theStart" targetRef="firstTask" />

    <userTask id="firstTask" />
    <sequenceFlow sourceRef="firstTask" targetRef="theEnd" />

    <endEvent id="theEnd" />
  </process>
```

```
    public class ExampleFieldInjectedExecutionListener implements ExecutionListener {

      private Expression fixedValue;

      private Expression dynamicValue;

      public void notify(ExecutionListenerExecution execution) throws Exception {
        execution.setVariable("var", fixedValue.getValue(execution).toString() + dynamicValue.getValue(execution).toStr
      }
    }
```

ExampleFieldInjectedExecutionListener 类串联了两个注入的属性。 （一个是固定的，一个是动态的）， 把他们保存到流程变量'var'中。

```
@Deployment(resources = {"org/activiti/examples/bpmn/executionListener/ExecutionListenersFieldInjectionProcess.bpmn20.x
public void testExecutionListenerFieldInjection() {
  Map<String, Object> variables = new HashMap<String, Object>();
  variables.put("myVar", "listening!");

  ProcessInstance processInstance = runtimeService.startProcessInstanceByKey("executionListenersProcess", variables);

  Object varSetByListener = runtimeService.getVariable(processInstance.getId(), "var");
  assertNotNull(varSetByListener);
  assertTrue(varSetByListener instanceof String);

  // Result is a concatenation of fixed injected field and injected expression
  assertEquals("Yes, I am listening!", varSetByListener);
}
```

Task listener 任务监听器

任务监听器可以在发生对应的任务相关事件时执行自定义 java 逻辑 或表达式。

任务监听器只能添加到流程定义中的用户任务中。 注意它必须定义在 BPMN 2.0 extensionElements 的子元素中， 并使用 activiti 命名空间，因为任务监听器是 activiti 独有的结构。

```
<userTask id="myTask" name="My Task" >
  <extensionElements>
    <activiti:taskListener event="create" class="org.activiti.MyTaskCreateListener" />
  </extensionElements>
</userTask>
```

任务监听器支持以下属性：

- event（必选）：任务监听器会被调用的任务类型。 可能的类型为：
  - create：任务创建并设置所有属性后触发。
  - assignment：任务分配给一些人时触发。 当流程到达userTask， assignment事件 会在create事件之前发生。 这样的顺序似乎不自然，但是原因很简单：当获得create时间时， 我们想获得任务的所有属性，包括执行人。
  - complete：当任务完成，并尚未从运行数据中删除时触发。
  - delete：只在任务删除之前发生。 注意在通过completeTask正常完成时，也会执行。
- class：必须调用的代理类。 这个类必须实现org.activiti.engine.delegate.TaskListener 接口。

  public class MyTaskCreateListener implements TaskListener {

  ```
  public void notify(DelegateTask delegateTask) {
    // Custom logic goes here
  }
  ```

  }

可以使用属性注入把流程变量或执行传递给代理类。注意代理类的实例是在部署时创建的（和 activiti 中其他类代理的情况一样），这意味着所有流程实例都会共享同一个实例。

- expression：（无法同时与 class 属性一起使用）： 指定事件发生时执行的表达式。 可以把 DelegateTask 对象和事件名称（使用 task.eventName ） 作为参数传递给调用的对象。

```
  <activiti:taskListener event="create" expression="${myObject.callMethod(task, task.eventName)}" />
```

- delegateExpression 可以指定一个表达式，解析一个实现了TaskListener 接口的对象， 这与服务任务一致。

```
  <activiti:taskListener event="create" delegateExpression="${myTaskListenerBean}" />
```

- 在 activiti 5.12 中，我们也介绍了新的任务监听 器，org.activiti.engine.impl.bpmn.listener.ScriptTaskListener。 脚本任务监听器可以为任务监听器事件执行脚本。

```
  <activiti:taskListener event="complete" class="org.activiti.engine.impl.bpmn.listener.ScriptTaskListener" >
    <activiti:field name="script">
      <activiti:string>
        def bar = "BAR";  // local variable
```

```
        foo = "FOO"; // pushes variable to execution context
        task.setOwner("kermit"); // test access to task instance
        bar // implicit return value
      </activiti:string>
    </activiti:field>
    <activiti:field name="language" stringValue="groovy" />
    <activiti:field name="resultVariable" stringValue="myVar" />
  <activiti:taskListener>
```

Multi-instance (for each) 多实例（循环）

Description 描述

多实例节点是在业务流程中定义重复环节的一个方法。 从开发角度讲，多实例和循环是一样的： 它可以根据给定的集合，为每个元素执行一个环节甚至一个完整的子流程， 既可以顺序依次执行也可以并发同步执行。

多实例是在一个普通的节点上添加了额外的属性定义 （所以叫做'多实例特性'），这样运行时节点就会执行多次。下面的节点都可以成为一个多实例节点

- User Task
- Script Task
- Java Service Task
- Web Service Task
- Business Rule Task
- Email Task
- Manual Task
- Receive Task
- (Embedded) Sub-Process
- Call Activity

网关和事件 不能设置多实例。

根据规范的要求，每个上级流程为每个实例创建分支时都要提供如下变量：

- nrOfInstances：实例总数
- nrOfActiveInstances：当前活动的，比如，还没完成的，实例数量。 对于顺序执行的多实例，值一直为1。
- nrOfCompletedInstances：已经完成实例的数目。
- 可以通过 execution.getVariable(x) 方法获得这些变量。 另外，每个创建的分支都会有分支级别的本地变量（比如，其他实例不可见， 不会保存到流程实例级别）：

- loopCounter：表示特定实例的在循环的索引值。可以使用 activiti 的 elementIndexVariable 属性修改 loopCounter 的变量名。

Graphical notation 图形标记

如果节点是多实例的，会在节点底部显示三条短线。 三条竖线表示实例会并行执行。 三条横线表示顺序执行。





XML representation 内容

要把一个节点设置为多实例，节点xml元素必须设置一个multiInstanceLoopCharacteristics 子元素

```
<multiInstanceLoopCharacteristics isSequential="false|true">
 ...
</multiInstanceLoopCharacteristics>
```

isSequential 属性表示节点是进行 顺序执行还是并行执行。实例的数量会在进入节点时计算一次。 有一些方法配置它。一种方法是使用 loopCardinality 子元素直接指定一个数字。

```
<multiInstanceLoopCharacteristics isSequential="false|true">
  <loopCardinality>5</loopCardinality>
</multiInstanceLoopCharacteristics>
```

也可以使用结果为整数的表达式：

```
<multiInstanceLoopCharacteristics isSequential="false|true">
  <loopCardinality>${nrOfOrders-nrOfCancellations}</loopCardinality>
</multiInstanceLoopCharacteristics>
```

另一个定义实例数目的方法是，通过 loopDataInputRef 子元素，设置一个类型为集合的流程变量名。 对于集合中的每个元素，都会创建一个实例。 也可以通过 inputDataItem 子元素指定集合。 下面的代码演示了这些配置：

```
<userTask id="miTasks" name="My Task ${loopCounter}" activiti:assignee="${assignee}">
  <multiInstanceLoopCharacteristics isSequential="false">
    <loopDataInputRef>assigneeList</loopDataInputRef>
    <inputDataItem name="assignee" />
  </multiInstanceLoopCharacteristics>
</userTask>
```

假设 assigneeList 变量包含这些值[kermit, gonzo, foziee]。 在上面代码中，三个用户任务会同时创建。每个分支都会拥有一个用名为assignee 的流程变量， 这个变量会包含集合中的对应元素，在例子中会用来设置用户任务的分配者。

loopDataInputRef 和 inputDataItem 的缺点是1）名字不好记， 2）根据 BPMN 2.0 格式定义，它们不能包含表达式。activiti通过在 multiInstanceCharacteristics 中设置 collection 和 elementVariable 属性解决了这个问题：

```
<userTask id="miTasks" name="My Task" activiti:assignee="${assignee}">
  <multiInstanceLoopCharacteristics isSequential="true"
     activiti:collection="${myService.resolveUsersForTask()}" activiti:elementVariable="assignee" >
  </multiInstanceLoopCharacteristics>
</userTask>
```

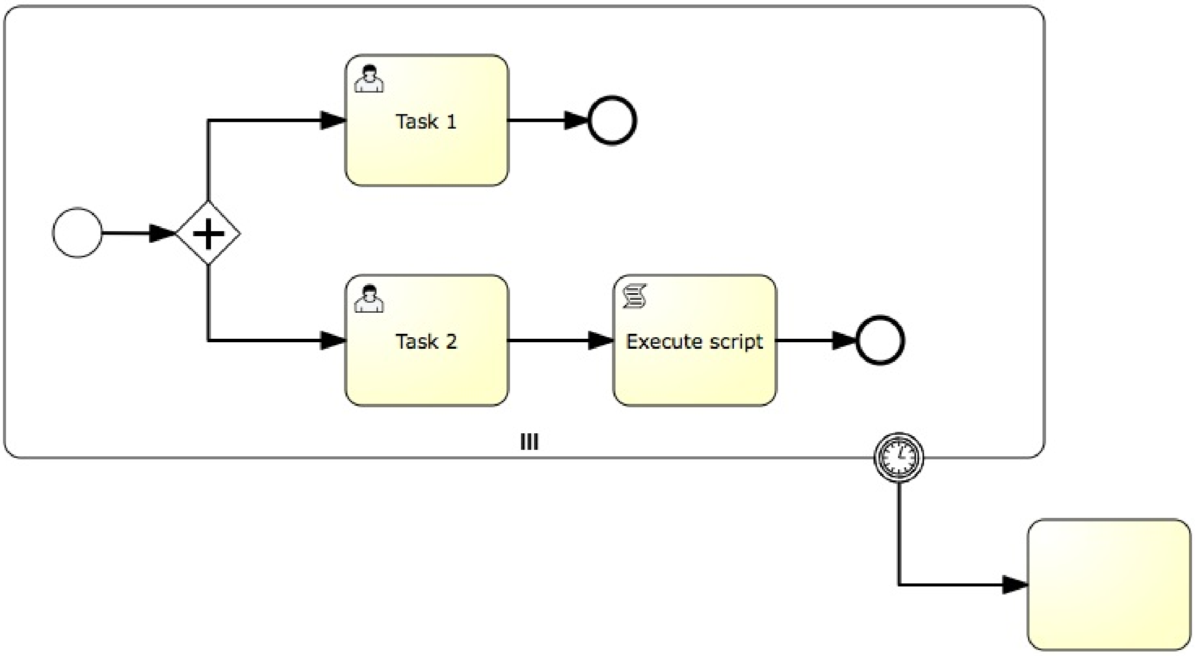
多实例节点在所有实例都完成时才会结束。也可以指定一个表达式在每个实例结束时执行。如果表达式返回true，所有其他的实例都会销毁，多实例节点也会结束，流程会继续执行。这个表达式必须定义在completionCondition 子元素中。

```
<userTask id="miTasks" name="My Task" activiti:assignee="${assignee}">
  <multiInstanceLoopCharacteristics isSequential="false"
     activiti:collection="assigneeList" activiti:elementVariable="assignee" >
    <completionCondition>${nrOfCompletedInstances/nrOfInstances >= 0.6 }</completionCondition>
  </multiInstanceLoopCharacteristics>
</userTask>
```

在这里例子中，会为 assigneeList 集合的每个元素创建一个并行的实例。 当60%的任务完成时，其他任务就会删除，流程继续执行。

Boundary events and multi-instance 边界事件和多实例

因为多实例是一个普通节点，它也可以在边缘使用边界事件。 对于中断型边界事件，当捕获事件时，所有激活的实例都会销毁。 参考以下多实例子流程



这里，子流程的所有实例都会在定时器触发时销毁，无论有多少实例， 也不管内部哪个节点没有完成。

Compensation Handlers 补偿处理器

Description 描述

[试验]

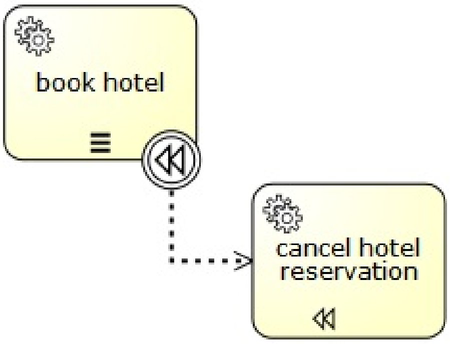
如果一个节点用来补偿另一个节点的业务，它可以声明为一个补偿处理器。 补偿处理器不包含普通的流，只在补偿事件触发时执行。

补偿处理器不能包含进入和外出顺序流。

补偿处理器必须使用直接关联分配给一个补偿边界事件。

Graphical notation 图形标记

如果节点是补偿处理器，补偿事件图标会显示在中间底部区域。下面的流程图显示了一个服务任务，附加了一个补偿边界事件， 并分配了一个补偿处理器。 注意 "cancel hotel reservation" 服务任务中间底部区域显示的补偿处理器图标。



XML representation 内容

为了声明作为补偿处理器的节点，我们需要把 isForCompensation 设置为 true：

```
<serviceTask id="undoBookHotel" isForCompensation="true" activiti:class="...">
</serviceTask>
```

Sub-Processes and Call Activities 子流程和调用节点

Sub-Process

Description 描述

子流程（Sub-process）是一个包含其他节点，网关，事件等等的节点。 它自己就是一个流程，同时是更大流程的一部分。 子流程是完全定义在父流程里的 （这就是为什么叫做内嵌子流程）。

子流程有两种主要场景：

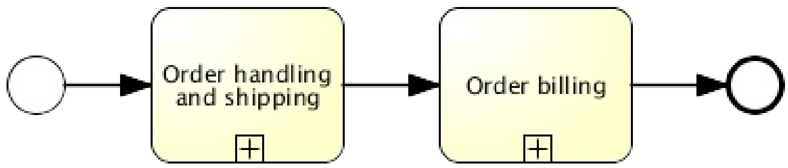
- 子流程可以使用继承式建模。 很多建模工具的子流程可以折叠， 把子流程的内部细节隐藏，显示一个高级别的端对端的业务流程总览。
- 子流程会创建一个新的事件作用域。 子流程运行过程中抛出的事件，可以被子流程边缘定义的 边界事件捕获， 这样就可以创建一个仅限于这个子流程的事件作用范围。

使用子流程要考虑如下限制：

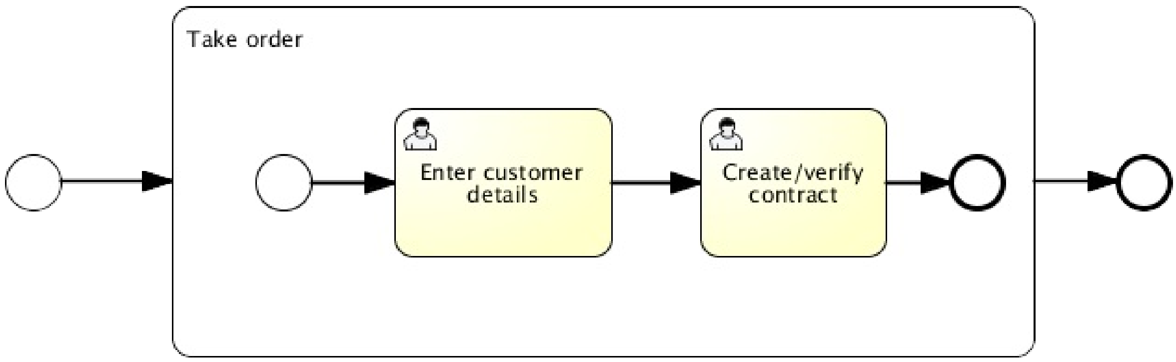
- 子流程只能包含一个空开始事件，不能使用其他类型的开始事件。子路程必须 至少有一个结束节点。注意，BPMN 2.0 规范允许忽略子流程的 开始和结束节点，但是当前 activiti 的实现并不支持。
- 顺序流不能跨越子流程的边界。

Graphical notation 图形标记

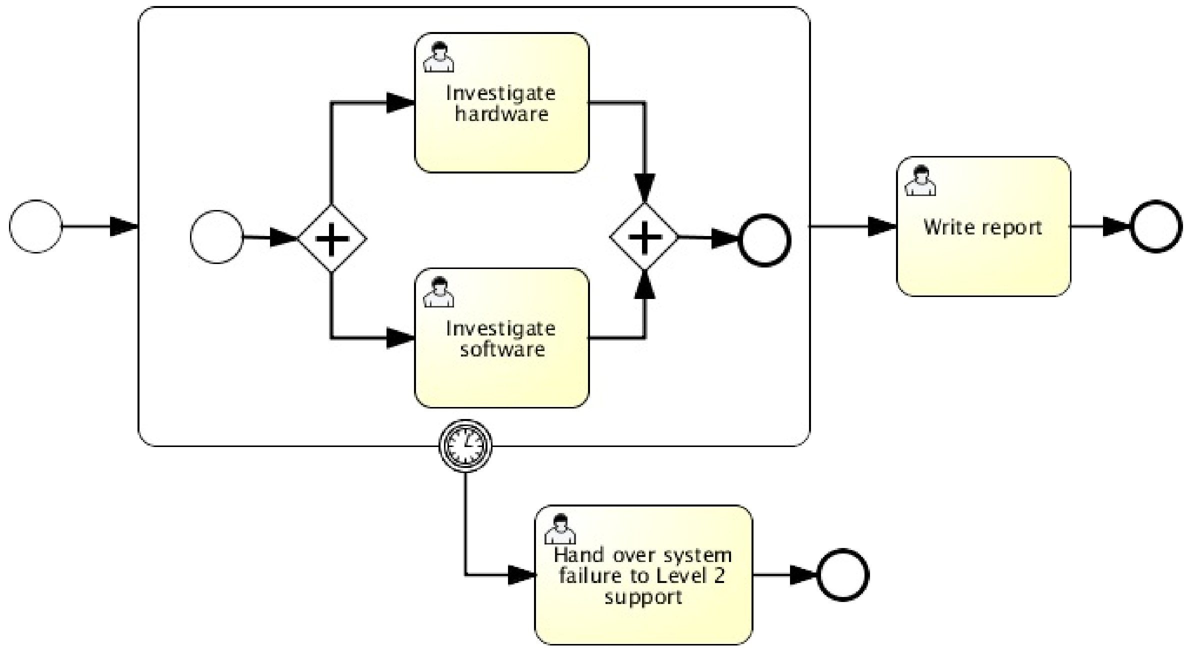
子流程显示为标准的节点，圆角矩形。 这时子流程是折叠的，只显示名称和一个加号标记，展示了高级别的流程总览：



这时子流程是展开的，子流程的步骤都显示在子流程边界内：



使用子流程的主要原因，是定义对应事件的作用域。下面流程模型演示了这个功能：调查软件/调查引荐任务需要同步执行，两个任务需要在同时完成，在二线支持解决之前。 这里，定时器的作用域（比如，节点需要及时完成）是由子流程限制的。



XML representation 内容

子流程定义为 subprocess 元素。 所有节点，网关，事件，等等。它是子流程的一部分，需要放在这个元素里。

```
<subProcess id="subProcess">

  <startEvent id="subProcessStart" />

  ... other Sub-Process elements ...

  <endEvent id="subProcessEnd" />

 </subProcess>
```

Event Sub-Process 事件子流程

Description 描述

事件子流程是 BPMN 2.0 中的新元素。事件子流程是由事件触发的子流程。 事件子流程可以添加到流程级别或任意子流程级别。用于触发事件子流程的事件是使用开始事件配置的。为此，事件子流程是不支持空开始事件的。 事件子流程可以被消息事件，错误事件，信号事件，定时器事件，或补偿事件触发。开始事件的订阅在包含事件子流程的作用域（流程实例 或子流程）创建时就会创建。 当作用域销毁也会删除订阅。

事件子流程可以是中断的或非中断的。一个中断的子流程会取消当前作用域内的所有流程。非中断事件子流程会创建那一个新的同步分支。中断事件子流程只会被每个激活状态的宿主触发一次， 非中断事件子流程可以触发多次。子流程是否是终端的，配置使用事件子流程的开始事件配置。

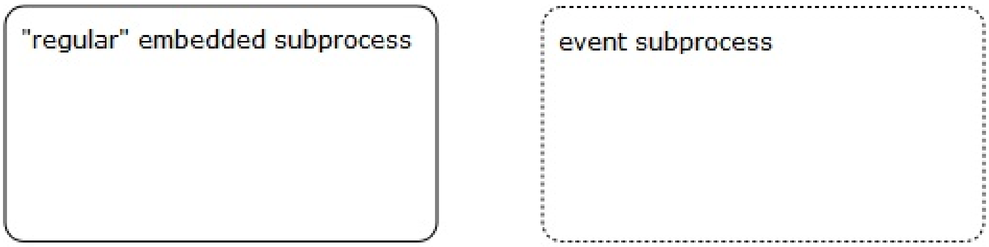
事件子流程不能有任何进入和外出流程。当事件触发一个事件子流程时，输入顺序流是没有意义的。 当事件子流程结束时，无论当前作用域已经结束了（中断事件子流程的情况），或为非中断子流程生成同步分支会结束。

当前的限制：

- activiti只支持中断事件子流程。
- activiti只支持使用错误开始事件或消息开始事件的事件子流程。

Graphical notation 图形标记

事件子流程可以显示为边框为虚线的内嵌子流程。



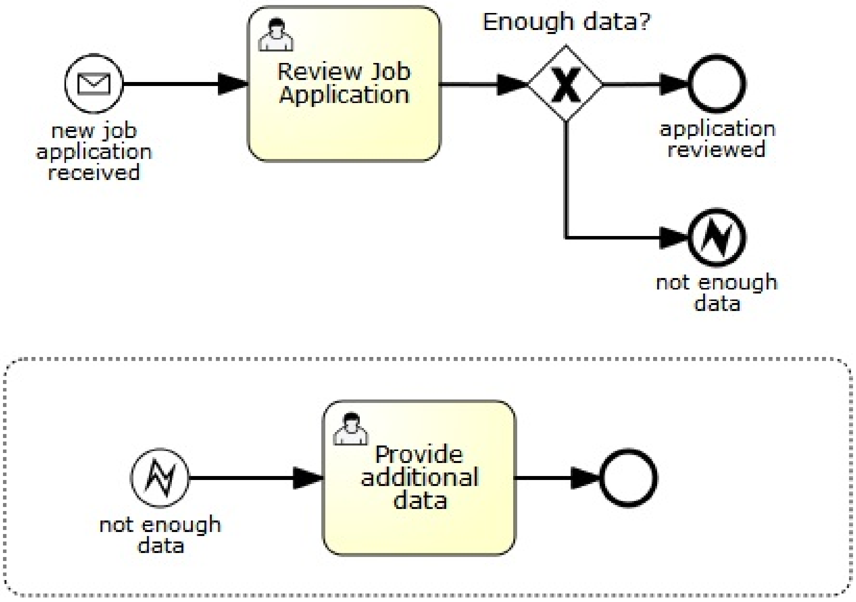
XML representation 内容

事件子流程的 XML 内容与内嵌子流程是一样的。 另外，要把triggeredByEvent 属性设置为 true：

```
<subProcess id="eventSubProcess" triggeredByEvent="true">
	...
</subProcess>
```

Example 实例

下面是一个使用错误开始事件触发的事件子流程的实例。事件子流程是放在“流程级别”的，意思是，作用于流程实例：

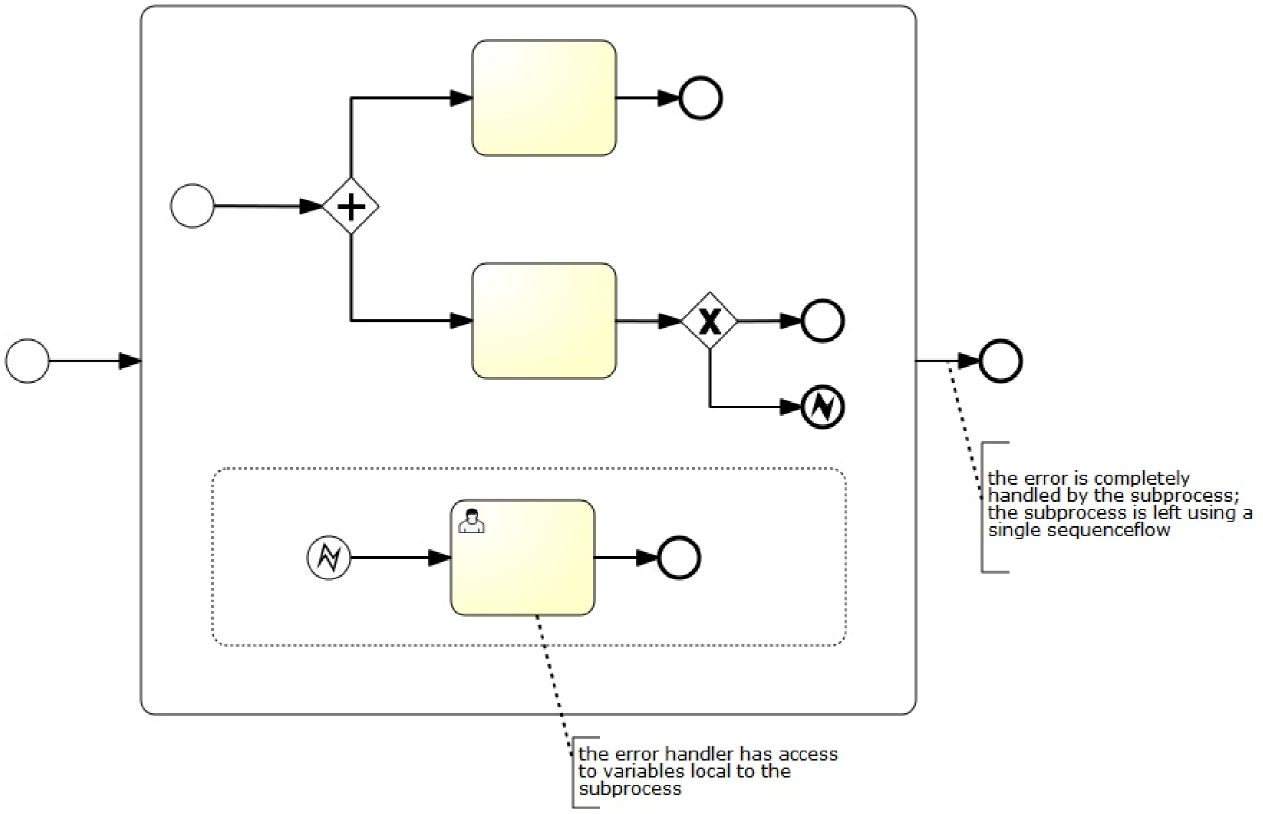


事件子流程的XML如下所示：

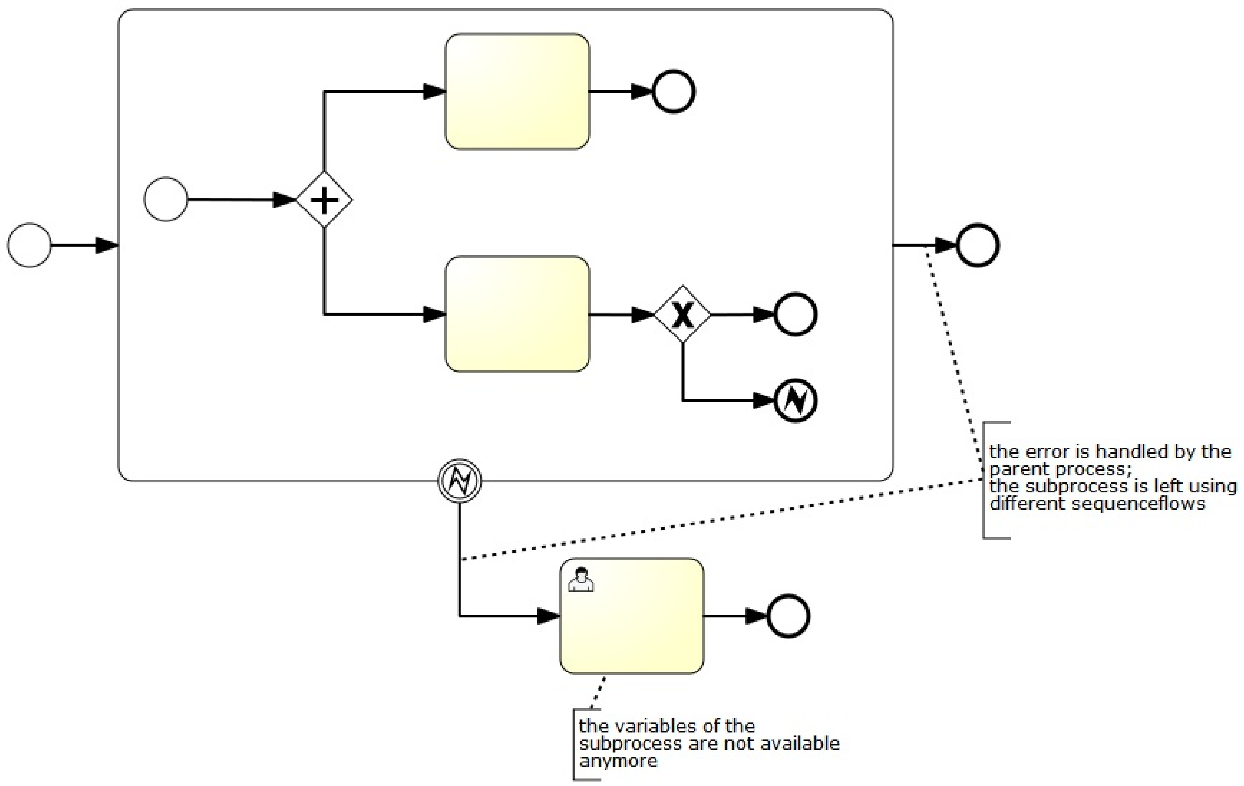
```
<subProcess id="eventSubProcess" triggeredByEvent="true">
	<startEvent id="catchError">
		<errorEventDefinition errorRef="error" />
	</startEvent>
	<sequenceFlow id="flow2" sourceRef="catchError" targetRef="taskAfterErrorCatch" />
	<userTask id="taskAfterErrorCatch" name="Provide additional data" />
```

```
</subProcess>
```

如上面所述，事件子流程也可以添加成内嵌子流程。如果添加为内嵌子流程，它其实是边界事件的一种替代方案。 考虑下面两个流程图。两种情况内嵌子流程会抛出一个错误事件。两种情况错误都会被捕获并使用一个用户任务处理。



相对于：



两种场景都会执行相同的任务。然而，两种建模的方式是不同的：

- 内嵌子流程是使用与执行作用域宿主相同的流程执行的。意思是内嵌子流程可以访问它作用域内的内部变量。 当使用边界事件时，执行内嵌子流程的流程会删除，并生成一个流程根据边界事件的顺序流继续执行。 这意味着内嵌子流程创建的变量不再起作用了。
- 当使用事件子流程时，事件是完全由它添加的子流程处理的。 当使用边界事件时，事件由父流程处理。

这两个不同点可以帮助我们决定是使用边界事件还是内嵌事件子流程来解决特定的流程建模/实现问题。

Transaction subprocess 事务子流程

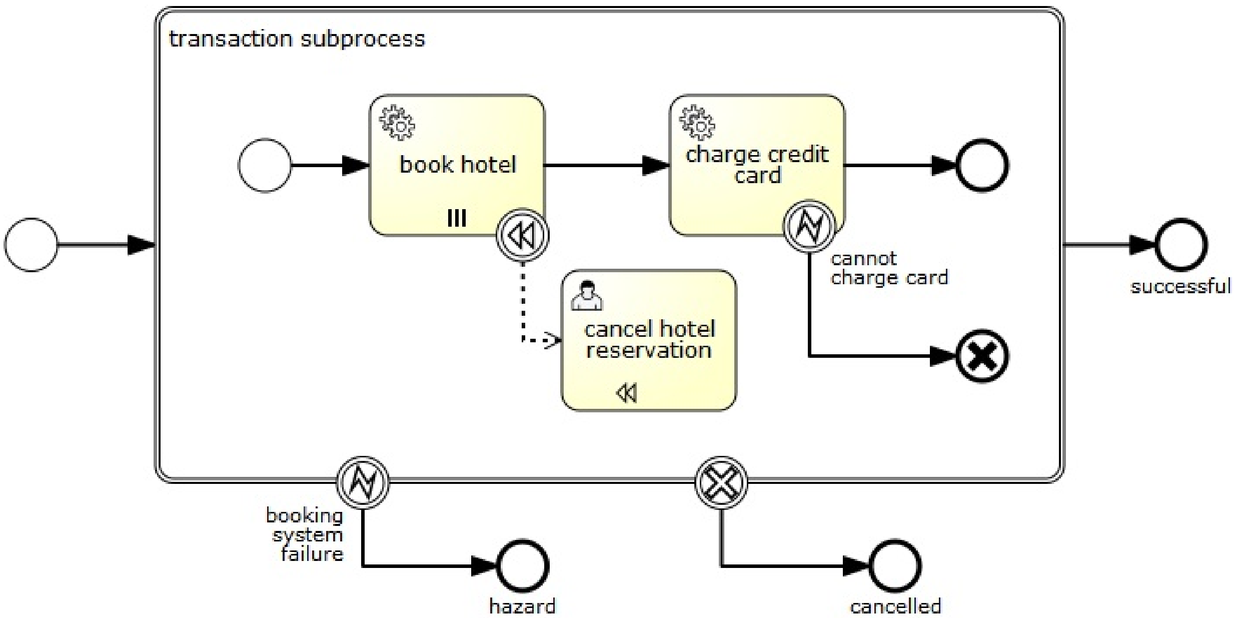
[试验]

Description 描述

事务子流程是内嵌子流程，可以用来把多个流程放到一个事务里。 事务是一个逻辑单元，可以把一些单独的节点放在一起，这样它们就可以一起成功或一起失败。事务可能的结果： 事务可以有三种可能的结果：

- 事务成功，如果没有取消也没有因为问题终结。如果事务子流程是成功的， 就会使用外出顺序流继续执行。 如果流程后来抛出了一个补偿事件，成功的事务可能被补偿。 注意：和普通内嵌子流程一样，事务可能在成功后， 使用中间补偿事件进行补偿。
- 事务取消，如果流程到达取消结束事件。这时，所有流程都会终结和删除。触发补偿的一个单独的流程，会通过取消边界事件继续执行。 在补偿完成之后，事务子流程会使用取消边界事务的外出顺序流向下执行。
- 事务被问题结束，如果跑出了一个错误事件， 而且没有在事务子流程中捕获。（如果错误被事务子流程的边界事件处理了，也会这样应用。） 这时，不会执行补偿。

下面的图形演示了三种不同的结果：



与**A CID** 事务的关系：一定不要把 bpmn 事务子流程与技术（ACID）事务相混淆。 bpmn事务子流程不是技术事务领域的东西。要理解 activiti 中的事务管理，请参考 并发与事务。 bpmn 事务和技术事务有以下不同点：

- ACID事务一般是短期的，bpmn事务可能持续几小时，几天，甚至几个月才能完成。（考虑事务中包含的节点可能有用户任务，一般人员响应的时间比应用时间要长。或者，在其他情况下，bpmn 事务可能要等待发生一些事务事件，就像要根据某种次序执行。） 这种操作通常要相比更新数据库的一条数据，或把一条信息保存到事务性队列中，消耗更长的时间来完成。
- 因为不能在整个业务节点的过程中保持一个技术性的事务，所以 bpmn 事务一般要跨越多个 ACID 事务。

- 因为 bpmn 事务会跨越多个 ACID 事务，所以会丧失 ACID 的特性。比如，考虑上述例子。 假设"约定旅店"和"刷信用卡"操作在单独的ACID事务中执行。 也假设"预定旅店"节点已经成功了。现在我们处于一个中间不稳定状态，因为我们预定了酒店，但是还没有刷信用卡。 现在，在一个ACID事务中，我们要依次执行不同的操作，也会有一个中间不稳定状态。 不同的是，这个中间状态对事务的外部是可见的。 比如，如果通过外部预定服务进行了预定，其他使用相同预定服务的部分就可以看到旅店被预定了。 这意味着实现业务事务时，我们完全失去了隔离属性 （注：我们也经常放弃隔离性，来为ACID事务获得更高的并发，但是我们可以完全控制，中间不稳定状态也只持续很短的时间）。

- bpmn 业务事务也不能使用通常的方式回滚。因为它跨越了多个事务，bpmn 事务取消时一些 ACID 事务可能已经提交了。 这时，它们不能被回滚了。

因为 bpmn 事务实际上运行时间很长，缺乏隔离性和回滚机制都需要被区别对待。 实际上，这里也没有更好的办法在特定领域处理这些问题：

- 使用补偿执行回滚。如果事务范围抛出了取消事件，会影响已经执行成功的节点， 并使用补偿处理器执行补偿。
- 隔离性的缺乏通常使用特定领域的解决方法来解决。比如，上面的例子中， 一个旅店房间可能会展示给第二个客户，在我们确认第一个客户付费之前。 虽然这可能与业务预期不符，预定服务可能选择允许一些过度的预约。
- 另外，因为事务会因为风险而中断，预定服务必须处理这种情况，已经预定了旅店，但是一直没有付款的情况。 （因为事务被中断了）。这时预定服务需要选择一个策略，在旅店房间预定超过最大允许时间后， 如果还没有付款，预定就会取消。

综上所述：ACID 处理的是通常问题（回滚，隔离级别和启发式结果）， 在实现业务事务时，我们需要找到特定领域的解决方案来处理这些问题。

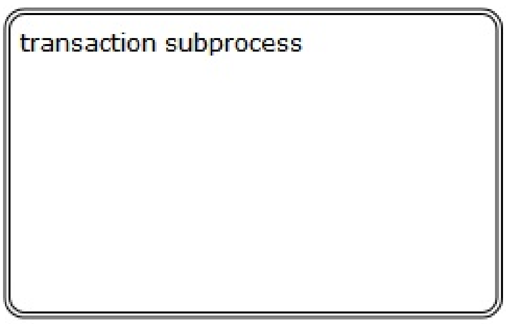
目前的限制：

- bpmn规范要求流程引擎能根据底层事务的协议处理事件，比如如果底层协议触发了取消事件，事务就会取消。 作为内嵌引擎，activiti目前不支持这项功能。（对此造成的后果，可以参考下面的一致性讨论）。

ACID事务顶层的一致性和优化并发： bpmn 事务保证一致性，要么所有节点都成功，或者一些节点成功，对其他成功的节点进行补偿。 无论哪种方式，都会有一致性的结果。不过要讨论一些 activiti 内部的情况，bpmn 事务的一致性模型是叠加在流程的一致性模型之上的。activiti 执行流程是事务性的。并发使用了乐观锁。在 activiti 中，bpmn 错误，取消和补偿事件都建立在同样的 acid 事务与乐观锁之上。 比如，取消结束事件只能触发它实际到达的补偿。如果之前服务任务抛出了未声明的异常。或者， 补偿处理器的效果无法提交，如果底层的 acid 事务的参与者把事务设置成必须回滚。 或者当两个并发流程到达了取消结束事件，可 能会触发两次补偿，并因为乐观锁异常失败。 所有这些都说明 activiti 中实现 bpmn 事务时，相同的规则也作用域普通的流程和子流程。 所以为了保证一致性，重要的是使用一种方式考虑实现乐观事务性的执行模型。

Graphical notation 图形标记

事务子流程显示为内嵌子流程，使用双线边框。



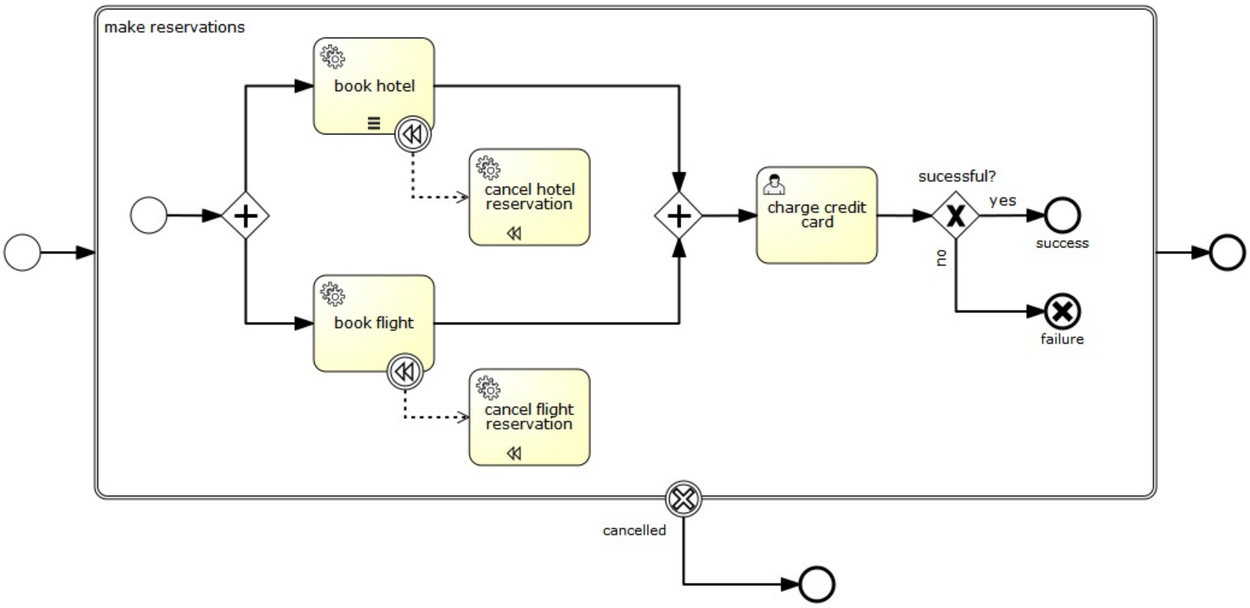
XML representation 内容

事务子流程使用 transaction 标签：

```
<transaction id="myTransaction" >
        ...
</transaction>
```

Example 实例

下面是事务子流程的实例：



Call activity (subprocess) 调用活动（子流程）

Description 描述

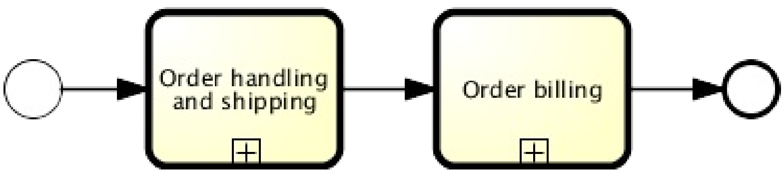
BPMN 2.0 区分了普通子流程， 也叫做内嵌子流程，和调用节点，看起来很相似。 从概念上讲，当流程抵达时， 两者都会调用子流程。

不同点是调用节点引用流程定义外部的一个流程，子流程 会内嵌到原始的流程定义中。使用调用节点的主要场景是需要重用流程定义， 这个流程定义需要被很多其他流程定义调用的时候。

当流程执行到调用节点，会创建一个新分支，它是到达调用节点的流程的分支。 这个分支会用来执行子流程，默认创建并行子流程，就像一个普通的流程。 上级流程会等待子流程完成，然后才会继续向下执行

Graphical notation 图形标记

调用节点显示与子流程相同， 不过是粗边框（无论是折叠和展开的）。 根据不同的建模工具，调用节点也可以展开，但是显示为折叠的子流程。



XML representation 内容

有种调用节点是经常性的节点，需要 calledElement 引用被 key 定义的流程。在实践中，这意味着流程的 ID 用于 calledElement。

```
<callActivity id="callCheckCreditProcess" name="Check credit" calledElement="checkCreditProcess" />
```

注意，子流程的流程定义是在执行阶段解析的。 就是说子流程可以与调用的流程分开部署，如果需要的话。

Passing variables 传递变量

可以把流程变量传递给子流程，反之亦然。数据会复制给子流程，当它启动的时候， 并在它结束的时候复制回主流程。

```
<callActivity id="callSubProcess" calledElement="checkCreditProcess" >
  <extensionElements>
          <activiti:in source="someVariableInMainProcess" target="nameOfVariableInSubProcess" />
          <activiti:out source="someVariableInSubProcss" target="nameOfVariableInMainProcess" />
  </extensionElements>
</callActivity>
```

我们使用 Activiti 扩展来简化 BPMN 标准元素调用dataInputAssociation 和 dataOutputAssociation，这只在你使用 BPMN 2.0 标准方式声明流程变量才管用。

这里也可以使用表达式：

```
<callActivity id="callSubProcess" calledElement="checkCreditProcess" >
        <extensionElements>
          <activiti:in sourceExpression="${x+5}"" target="y" />
          <activiti:out source="${y+5}" target="z" />
        </extensionElements>
</callActivity>
```

最后 z = y + 5 = x + 5 + 5

Example 实例

下面的流程图演示了简单订单处理。先判断客户端信用，这可能与很多其他流程相同。 检查信用阶段这里设计成调用节点。



流程看起来像这样的：

```
<startEvent id="theStart" />
<sequenceFlow id="flow1" sourceRef="theStart" targetRef="receiveOrder" />

<manualTask id="receiveOrder" name="Receive Order" />
<sequenceFlow id="flow2" sourceRef="receiveOrder" targetRef="callCheckCreditProcess" />

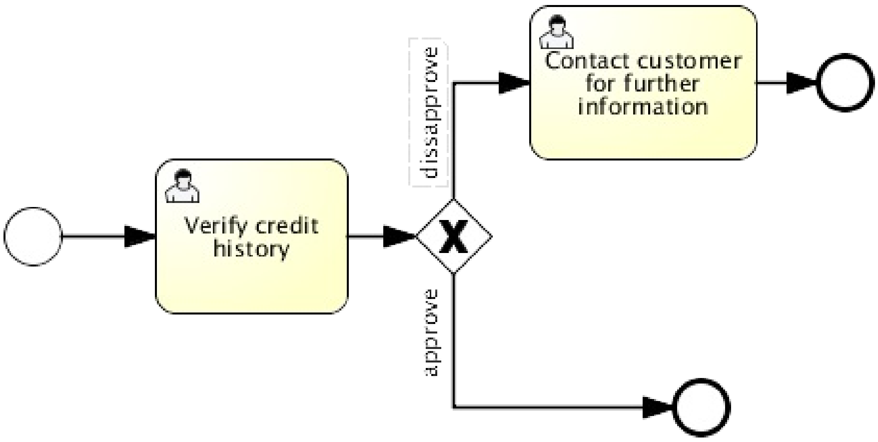
<callActivity id="callCheckCreditProcess" name="Check credit" calledElement="checkCreditProcess" />
<sequenceFlow id="flow3" sourceRef="callCheckCreditProcess" targetRef="prepareAndShipTask" />

<userTask id="prepareAndShipTask" name="Prepare and Ship" />
```

```
<sequenceFlow id="flow4" sourceRef="prepareAndShipTask" targetRef="end" />

<endEvent id="end" />
```

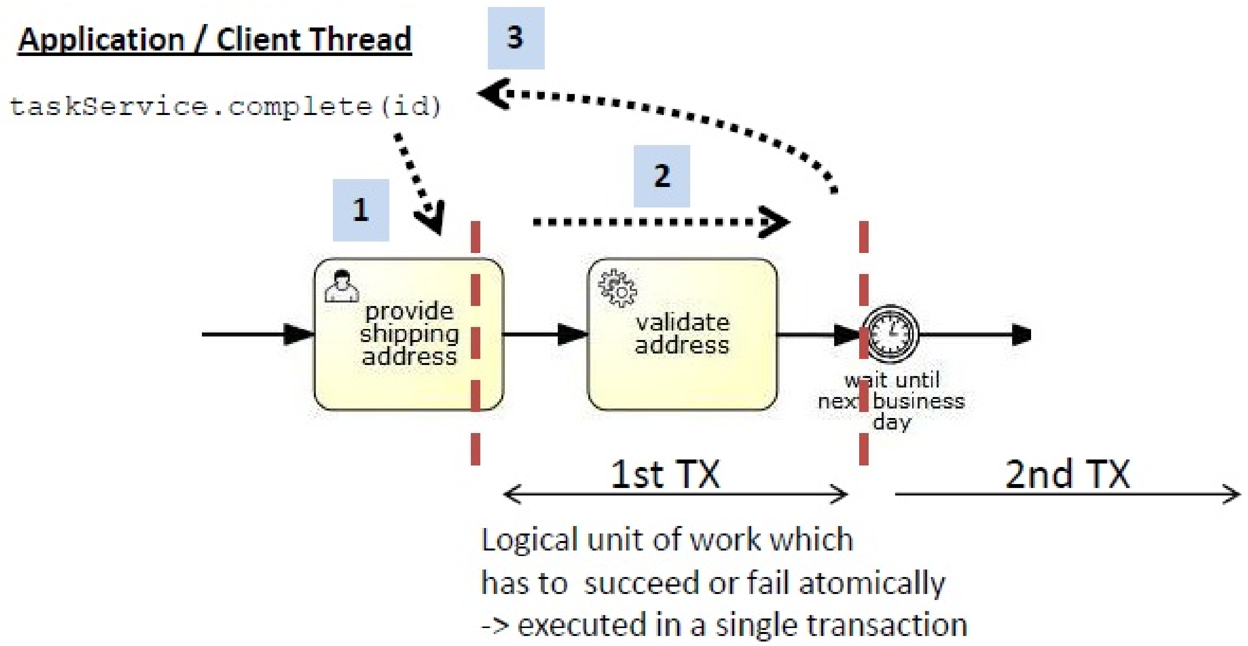
子流程看起来像这样的：



Transactions and Concurrency 事务与并发

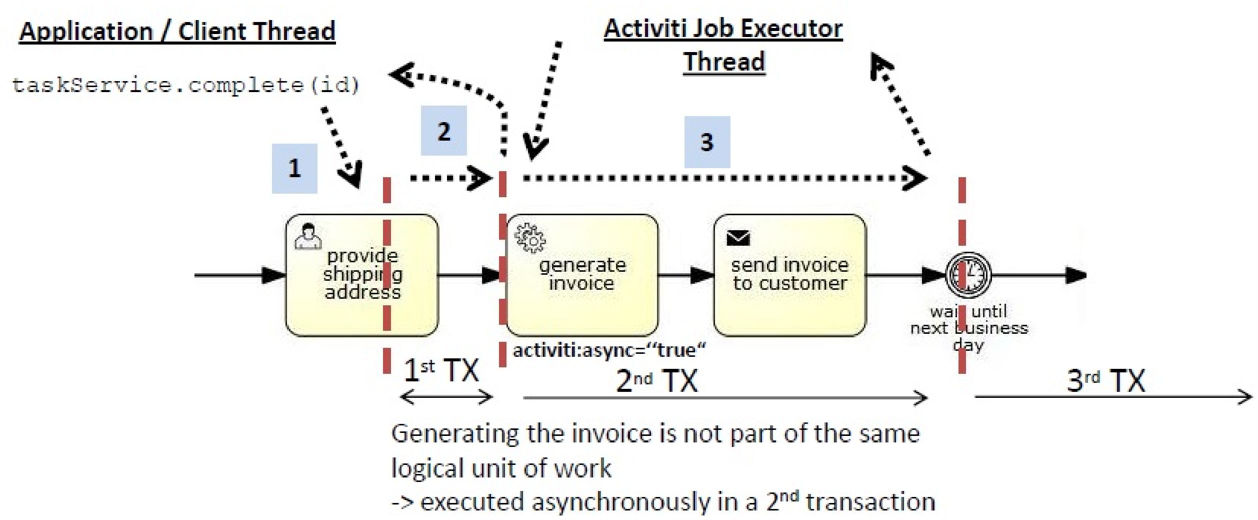
Asynchronous Continuations 异步操作

Activiti 通过事务方式执行流程，可以根据你的需求定制。现在开始看一下 Activiti 通常是如何处理事务的。 如果触发了 Activiti 的操作（比如，开始流程，完成任务，触发流程继续执行）， Activiti 会推进流程，直到每个分支都进入等待状态。 更抽象的说，它会流程图执行深度优先搜索， 如果每个分支都遇到等待状态，就会返回。等待状态是"稍后"需要执行任务，就是说 Activiti 会把当前状态保存到数据库中，然后等待下一次触发。 触发可能来自外部，比如用户任务或接收到一个消息，也可能来自 Activiti 本身，比如我们设置了定时器事件。 下面图片展示了这种操作：



我们可以看到包含用户任务，服务任务和定时器事件的流程。完成用户任务，和校验地址是在同一个工作单元中， 所以它们的成功和失败是原子性的。意味着如果服务任务抛出异常，我们要回滚当前事务， 这样流程会退回到用户任务，用户任务就依然在数据库里。 这就是 Activiti 默认的行为。在（1）中应用或客户端线程完成任务。这会执行服务，流程推进，直到遇到一个等待状态， 这里就是定时器（2）。然后它会返回给调用者（3），并提交事务（如果事务是由activiti开启的）。

有的时候，这不是我们想要的。有时我们需要自己控制流程中事务的边界，这样就能把业务逻辑包裹在一起。 这就需要使用异步执行了。参考下面的流程（判断）：



这次我们完成了用户任务，生成一个发票，把发票发送给客户。 这次生成发票不在同一个工作单元内了，所以我们不想对用户任务进行回滚，如果生成发票出错了。 所以，我们想让 Activiti 实现的是完成用户任务（1），提交事务，返回给调用者应用。然后在后台的线程中，异步执行生成发票。 后台线程就是 Activiti 的 job 执行器（其实是一个线程池）周期对数据库的 job 进行扫描。 所以后面的场景，当我们到达"generate invoice"任务，我们为 activiti 创建一个稍后执行的job "消息"， 并把它保存到数据库。job 会被 job 执行器获取并执行。我们也会给本地 job 执行器一个提醒，告诉它有一个新 job，来增加性能。

要想使用这个特性，我们要使用 activiti:async="true" 扩展。例子中，服务任务看起来就是这样：

```
<serviceTask id="service1" name="Generate Invoice" activiti:class="my.custom.Delegate" activiti:async="true" />
```

activiti:async 可以使用到如下 bpmn 任务类型中： task, serviceTask, scriptTask, businessRuleTask, sendTask, receiveTask, userTask, subProcess, callActivity

对于 userTask，receiveTask 和其他等待装填，异步执行的作用是让开始流程监听器运行在一个单独的线程/事务中。

Fail Retry 失败重试

Activiti 在其默认配置，重试 3 次工作，当在一个作业执行遇到任何异常情况。这对异步任务工作来说也成立。在某些情况下，就需要更多的灵活性。有两个参数进行配置：

- 重试次数
- 重试延迟时间

这些参数可以通过配置 activiti:failedJobRetryTimeCycle。这里是一个简单的使用示例：

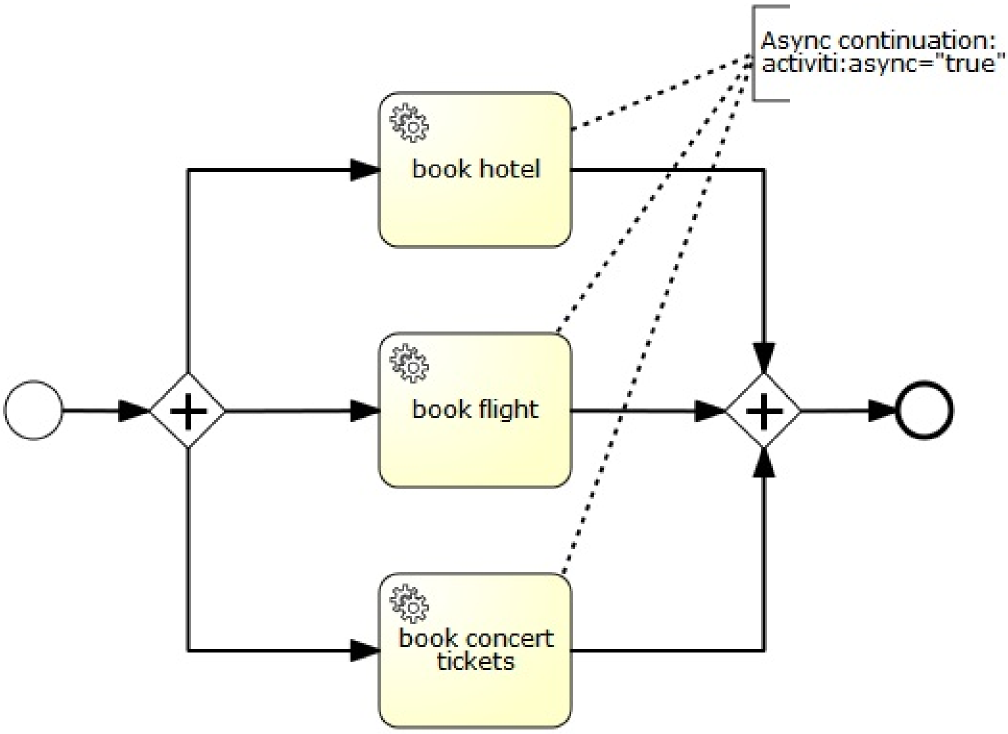
```
<serviceTask id="failingServiceTask" activiti:async="true" activiti:class="org.activiti.engine.test.jobexecutor.RetryFa
        <extensionElements>
                <activiti:failedJobRetryTimeCycle>R5/PT7M</activiti:failedJobRetryTimeCycle>
        </extensionElements>
</serviceTask>
```

Exclusive Jobs 排他任务

从 Activiti 5.9开始，JobExecutor 能保证同一个流程实例中的 job 不会并发执行。为啥呢？

Why exclusive Jobs?为什么要使用排他任务？

参考如下流程定义：



我们有一个并行网关，后面有三个服务任务，它们都设置为异步执行。这样会添加三个 job 到数据库里。 一旦 job 进入数据库，它就可以被jobExecutor 执行了。JobExecutor 会获取 job，把它们代理到工作线程的线程池中，会在那里真正执行 job。 就是说，使用异步执行，你可以把任务分配给这个线程池（在集群环境，可能会使用多个线程池）。这通常是个好事情。 然而它也会产生问题：一致性。考虑一下服务任务后的汇聚。 当服务任务完成后，我们到达并发汇聚节点，需要决定是等待其他分支，还是继续向下执行。 就是说，对每个到达并行汇聚的分支，我们都需要判断是继续还是等待其他分支的一个或多个分支。

为什么这就是问题了呢？因为服务任务配置成使用异步执行，可能相关的job 都在同一时间被获取，被 JobExecutor 分配给不同的工作线程执行。 结果是三个单独的服务执行使用的事务在到达并发汇聚时可能重叠。如果出现了这个问题，这些事务是互相不可见的， 其他事务同时到达了相同的并发汇聚，假设它们都在等待其他分支。然而，每个事务都假设它们在等待其他分支， 所以没有分支会越过并发汇聚继续执行，流程实例会一直在等待状态，无法继续执行。

Activiti 是如何解决这个问题的？ Activiti 使用了乐观锁。当我们基于判断的数据看起来不是最新的时 （因为其他事务可能在我们提交之前进行了修改，我们会在每个事务里增加数据库同一行的版本）。这时，第一个提交的事务会成功， 其他会因为乐观锁异常导致失败。这就解决了我们上面讨论的流程的问题：如果多个分支同步到达并行汇聚， 它们会假设它们都在登录，并增加它们父流程的版本号（流程实例）然后尝试提交。 第一个分支会成功提交，其他分支会因为乐观锁导致失败。因为流程是被 job 触发的， Activiti 会尝试在等待一段时间后尝试执行同一个 job，想这段时间可以同步网关的状态。

这是一个很好的解决方案吗？像我们看到的一样，乐观锁允许 Activiti 避免非一致性。它确定我们不会“堵在汇聚网关”， 意思是：或者所有分支都通过网关，或者数据库中的 job 正在尝试通过。然而， 虽然这是一个对于持久性和一致性的完美解决方案， 但对于上层来说不一定是期望的行为：

- Activiti 只会对同一个 job 重试估计次数（默认配置为3）。之后，job 还会在数据库里，但是不会再重试了。 意味着这个操作必须手工执行job 的触发。
- 如果 job 有非事务方面的效果，它不会因为失败的事务回滚。比如，如果“预定演唱会门票”服务没有与 Activiti 共享事务， 重试 job 可能导致我们预定了过多门票。

在 Activiti 5.9 中，我们推荐了新的概念，并已经在 jBPM 4中实现了，叫做“排他job”。

What are exclusive jobs? 什么是排他job？

对于一个流程实例，排他任务不能同时执行两个。考虑上面的流程： 如果我们把服务任务申请为排他任务，JobExecutor 会

保证对应的 job 不会并发执行。 相反，它会保证无论什么时候获取一个流程实例的排他任务，都会把同一个流程实例的其他任务都取出来，放在同一个工作线程中执行。 它保证 job 是顺序执行的。

如何启用这个特性？从 Activiti 5.9 开始，排他任务已经是默认配置了。所以异步执行和定时器事件默认都是排他任务。 另外，如果你想把 job 设置为非排他，可以使用 activiti:exclusive="false" 进行配置。 比如，下面的服务任务就是异步但是非排他的。

```
<serviceTask id="service" activiti:expression="${myService.performBooking(hotel, dates)}" activiti:async="true" activit
```

这是一个好方案吗？ 有一些人问我们这是否是一个好方案。他们的结论会帮你在并发和性能问题方面节省时间。 这个问题上需要考虑两件事情：

- 如果是你是专家并且知道自己在做什么时（理解“为什么排他任务”这章的内容）， 也可以关闭这个功能， 否则，对于大多数使用异步执行和定时器的用户来说，这个功能是没问题的。
- 它也没有性能问题，在高负载的情况下性能是个问题。高负载意味着JobExecutor 的所有工作线程都一直在忙碌着。 使用排他任务，Activiti 可以简单的分布不同的负载。排他任务意味着同一个流程实例的异步执行会由相同的线程顺序执行。 但是要考虑：如果你有多个流程实例时。 所有其他流程实例的 job 也会分配给其他线程同步执行。 意味着虽然 Activiti 不会同时执行一个流程实例的排他 job，但是还会同步执行多个流程实例的一步执行。 通过一个总体的预测，在大多数场景下，它都会让单独的实例运行的更迅速。 而且，对于同一流程实例中的 job，需要用到的数据也会利用执行的集群节点的缓存。 如果任务没有在同一个节点执行，数据就必须每次从数据库重新读取了

Process Initiation Authorization 流程起始授权

默认所有人在部署的流程定义上启动一个新流程实例。通过流程初始化授权功能定义的用户和组，web 客户端可以限制哪些用户可以启动一个新流程实例。 注意：Activiti 引擎不会校验授权定义。 这个功能只是为减轻 web 客户端开发者实现校验规则的难度。 设置方法与用户任务用户分配类似。 用户或组可以使用 标签分配为流程的默 认启动者。下面是一个例子：

```
<process id="potentialStarter">
     <extensionElements>
       <activiti:potentialStarter>
         <resourceAssignmentExpression>
           <formalExpression>group2, group(group3), user(user3)</formalExpression>
         </resourceAssignmentExpression>
       </activiti:potentialStarter>
     </extensionElements>
   <startEvent id="theStart"/>
   ...
```

上面的 XML 中，user(user3) 是直接引用了用户 user3，group(group3) 是引用了组 group3。如果没显示设置，默认认为是群组。 也可以使用 标签的属性， 和。 下面是一个例子：

```
<process id="potentialStarter" activiti:candidateStarterUsers="user1, user2"
                                        activiti:candidateStarterGroups="group1">
      ...
```

可以同时使用这两个属性。

定义流程初始化授权后，开发者可以使用如下方法获得授权定义。 这些代码可以获得给定的用户可以启动哪些流程定义：

```
processDefinitions = repositoryService.createProcessDefinitionQuery().startableByUser("userxxx").list();
```

也可以获得指定流程定义设置的潜在启动者对应的 identity link。

```
identityLinks = repositoryService.getIdentityLinksForProcessDefinition("processDefinitionId");
```

下面例子演示了如何获得可以启动给定流程的用户列表：

```
List<User> authorizedUsers =  identityService().createUserQuery().potentialStarter("processDefinitionId").list();
```

相同的方式，获得可以启动给定流程配置的群组：

```
List<Group> authorizedGroups =  identityService().createGroupQuery().potentialStarter("processDefinitionId").list();
```

Data objects 数据对象

[试验]

BPMN 提供了一种功能，可以在流程定义或子流程中定义数据对象。根据BPMN 规范，流程定义可以包含复杂 XML 结构，可以导入 XSD 定义。对于 Activiti 来说，作为 Activiti 首次支持的数据对象，可以支持如下的 XSD 类型：

```
<dataObject id="dObj1" name="StringTest" itemSubjectRef="xsd:string"/>
```

```
<dataObject id="dObj2" name="BooleanTest" itemSubjectRef="xsd:boolean"/>
```

```
<dataObject id="dObj3" name="DateTest" itemSubjectRef="xsd:datetime"/>
```

```
<dataObject id="dObj4" name="DoubleTest" itemSubjectRef="xsd:double"/>
```

```
<dataObject id="dObj5" name="IntegerTest" itemSubjectRef="xsd:int"/>
```

```
<dataObject id="dObj6" name="LongTest" itemSubjectRef="xsd:long"/>
```

数据对象定义会自动转换为流程变量，名称与 'name' 属性对应。 除了数据对象的定义之外，Activiti 也支持使用扩展元素来为这个变量赋予默认值。下面的 BPMN 片段就是对应的例子：

```
<process id="dataObjectScope" name="Data Object Scope" isExecutable="true">
          <dataObject id="dObj123" name="StringTest123" itemSubjectRef="xsd:string">
            <extensionElements>
              <activiti:value>Testing123</activiti:value>
            </extensionElements>
          </dataObject>
```

Chapter 9. Forms 表单

Activiti 提供了一种方便而且灵活的方式在业务流程中以手工方式添加表单。我们对表单的 支持有2种方式： 通过表单属性对内置表单进行渲染和外置表单进行渲染。

Form properties 表单属性

业务流程相关联的所有信息要么是包含自身的流程变量，要么是通过流程变量的引用。Activiti 支持存储复杂的 Java 对象作为流程变量，如 Serializable（序列化）对象， Jpa 实体对象或者整个 XML 文档作为字符串。

用户是在启动一个流程和完成用户任务时，与流程进行交互的。表单需要某个 UI 技术渲染之后才能够与用户进行交互。为了能够使用不同 UI 技术变得容易，流程定义包含一个对流程变量中复杂的 Java 类型对象到一个 `properties` 的 `Map<String,String>` 类型的转换逻辑。

使用 Activiti API 的方法查看公开的属性信息。然后，任意 UI 技术都能够在这些属性上面构建一个表单。该属性专门（并且更多局限性）为流程变量提供了一个视图。 表单所需要显示的属性可以从下面例子中的返回值FormData 中获取。

```
StartFormData FormService.getStartFormData(String processDefinitionId)
```

或者

```
TaskFormdata FormService.getTaskFormData(String taskId)
```

在默认情况下，内置的表单引擎，这些属性就像对流程变量一样。如果任务表单属性和流程变量是一对一的关系，那么任务表单属性就不需要进行申明了，例如，下面的申明：

```
<startEvent id="start" />
```

当执行到开始事件时，所有的流程变量都是可用的，但

```
formService.getStartFormData(String processDefinitionId).getFormProperties()
```

会是一个空值，因为没有定义具体的映射。

在上面的实例中，所有被提交的属性都将会作为流程变量被存储在Activiti 使用的数据库中。这意味着在一个表单中新添加一个简单的input 输入字段，也会作为一个新的变量被存储。

属性来自于流程变量，但不一定非要作为流程变量存储起来，例如，一个流程变量可能是 JPA 实体如类 Address。在某种 UI 技术中使用的表单属性StreetName可能会关联到一个表达式 `#{address.street}` 。

类似的，用户提交的表单属性应该作为流程变量进行存储或者使用 UEL 值表达式将其作为流程变量的一个嵌套属性进行存储，例如 `#{address.street}` 。

同样的，提交的表单属性默认的行为是作为流程变量进行存储，除非一个 `formProperty` 申明了其他的规则。

类型转换同样也可以应用于表单数据和流程变量之间处理的一部分。

举例

```
<userTask id="task">
  <extensionElements>
    <activiti:formProperty id="room" />
    <activiti:formProperty id="duration" type="long"/>
    <activiti:formProperty id="speaker" variable="SpeakerName" writable="false" />
```

```
    <activiti:formProperty id="street" expression="#{address.street}" required="true" />
  </extensionElements>
</userTask>
```

- 表单属性 room 将会被映射为 String 类型流程变量 room。
- 表单属性 duration 将会被映射为 java.lang.Long 类型流程变量 duration。
- 表单属性 speaker 将会被映射为流程变量 SpeakerName。它只能够在TaskFormData 对象中使用。如果 属性 speaker 提交，将会抛出一个ActivitiException 的异常。类似的，将其属性设置为readable="false",该属性就会在 FormData 进行排除，但是在提交后仍然会对其进行处理。
- 表单属性 street 将会映射为 Java Bean address 的属性 street 作为 String 类型的流程变量。 当提交的表单属性并没有提供并且 required="true" 时，那么就会抛出一个异常。

表单数据也可以作为 FormData 的一部分提供类型元数据。该 FormData可以从 StartFormData FormService.getStartFormData(String processDefinitionId)和TaskFormdata FormService.getTaskFormData(String taskId)方法的返回值中获取。

我们支持以下的几种表单属性类型：

- string (org.activiti.engine.impl.form.StringFormType
- long (org.activiti.engine.impl.form.LongFormType)
- enum (org.activiti.engine.impl.form.EnumFormType)
- date (org.activiti.engine.impl.form.DateFormType)
- boolean (org.activiti.engine.impl.form.BooleanFormType)

对于申明每一个表单属性，以下的 FormProperty 信息可以通过List formService.getStartFormData(String processDefinitionId).getFormProperties() 和 List formService.getTaskFormData(String taskId).getFormProperties() 获取。

```
public interface FormProperty {
  /** the key used to submit the property in {@link FormService#submitStartFormData(String, java.util.Map)}
   * or {@link FormService#submitTaskFormData(String, java.util.Map)} */
  String getId();
  /** the display label */
  String getName();
  /** one of the types defined in this interface like e.g. {@link #TYPE_STRING} */
  FormType getType();
  /** optional value that should be used to display in this property */
  String getValue();
  /** is this property read to be displayed in the form and made accessible with the methods
   * {@link FormService#getStartFormData(String)} and {@link FormService#getTaskFormData(String)}. */
  boolean isReadable();
  /** is this property expected when a user submits the form? */
  boolean isWritable();
  /** is this property a required input field */
  boolean isRequired();
}
```

举例

```
<startEvent id="start">
  <extensionElements>
    <activiti:formProperty id="speaker"
      name="Speaker"
      variable="SpeakerName"
      type="string" />

    <activiti:formProperty id="start"
      type="date"
      datePattern="dd-MMM-yyyy" />

    <activiti:formProperty id="direction" type="enum">
      <activiti:value id="left" name="Go Left" />
      <activiti:value id="right" name="Go Right" />
```

```
        <activiti:value id="up" name="Go Up" />
        <activiti:value id="down" name="Go Down" />
      </activiti:formProperty>

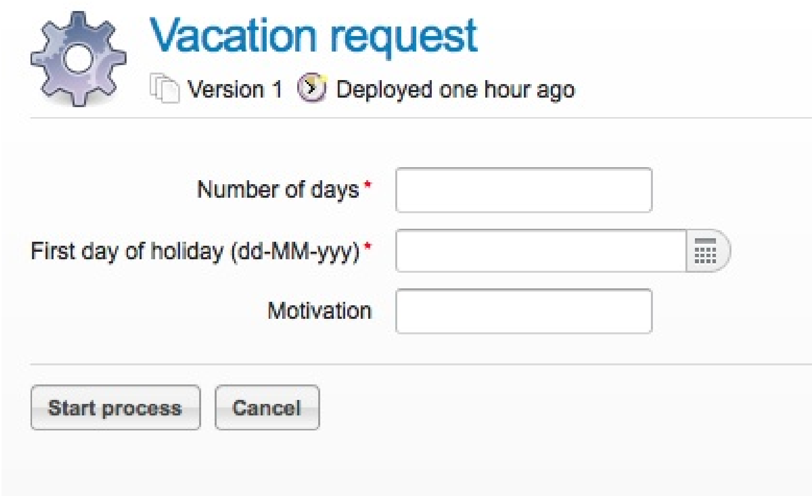
    </extensionElements>
  </startEvent>
```

所有的表单属性的信息都是可以通过 API 进行访问的。可以通过 formProperty.getType().getName() 获取类型的名称。 甚至可以通过 formProperty.getType().getInformation("datePattern") 获取日期的匹配方式。通过 formProperty.getType().getInformation("values") 可以获取到枚举值。

Activiti 控制台支持表单属性并且可以根据表单定义对表单进行渲染。例如下面的 XML 片段

```
  <startEvent>
    <extensionElements>
      <activiti:formProperty id="numberOfDays" name="Number of days" value="${numberOfDays}" type="long" required="true"/
      <activiti:formProperty id="startDate" name="First day of holiday (dd-MM-yyy)" value="${startDate}" datePattern="dd-
      <activiti:formProperty id="vacationMotivation" name="Motivation" value="${vacationMotivation}" type="string" />
    </extensionElements>
  </userTask>
```

当使用 Activiti 控制台时，它将会被渲染成流程的启动表单。



External form rendering 外部表单的渲染

该 API 同样也允许你执行 Activiti 流程引擎之外的方式渲染你自己的任务表单。这些步骤说明你可以用你自己的方式对任务表单进行渲染。

本质上，所有需要渲染的表单属性都是通过2个服务方法中的一个进行装配的： StartFormData FormService.getStartFormData(String processDefinitionId) 和 TaskFormdata FormService.getTaskFormData(String taskId)

表单属性可以通过 ProcessInstance FormService.submitStartFormData(String processDefinitionId,Map properties) 和 void FormService.submitStartFormData(String taskId,Map properties)2种方式进行提交。

想要了解更多表单属性如何映射为流程变量，可以查看 第 9.1 节 “表单属性”

你可以将任何表单模版资源放进你要部署的业务文档之中（如果你想要将它们按照流程的版本进行存储）。它将会在部署中作为一种可用的资源。你可以使用String ProcessDefinition.getDeploymentId() 和 InputStream RepositoryService.getResourceAsStream(String deploymentId, String resourceName); 获取部署上去的表单模版。这样就可以获取你的表单模版定义文件， 那么你就可以在你自己的应用中渲染/显示你的表单。

你也可以使用该功能获取任务表单之外的其他的部署资源用于其他的目的。 属性 `<userTask activiti:formKey="..."` 通过 API String FormService.getStartFormData(String processDefinitionId).getFormKey() 和 String FormService.getTaskFormData(String taskId).getFormKey()暴 露出来的。 你可以使用这个存储你部署的模版中的全名（例如 `org/activiti/example/form/mycustom-form.xml` ）,但是这并不是必须的。 例如，你可以在表单属性中存储一个通用的 key，然后运用一种算法或者换转去得到你实际使用的模版。当你想要通过不同UI技术渲染不能的表 单，这可能更加方便，例如，使用正常屏幕大小的web应用程序的表单，移动手机小屏幕的表单和甚至可能是 IM 表单和 email 表单模版。

Chapter 10. JPA

你可以使用JPA实体作为流程变量，并且可以这样做：

- 基于流程变量更新已有的JPA实体，它可以在用户任务的表单中填写或者由服务任务生成。
- 重用已有的领域模型不需要编写显示的服务获取实体或者更新实体的值。
- 根据已有实体的属性做出判断（网关即分支聚合）。
- ...

10.1. Requirements 要求

只支持符合以下要求的实体：

- 实体应该使用 JPA 注解进行配置，我们支持字段和属性访问两种方式。映射的超级累类也能够被使用。
- 实体中应该有一个使用 `@Id` 注解的主键，不支持复合主键 (`@EmbeddedId` 和 `@IdClass`)。Id 字段/属性能够使用 JPA 规范支持的任意类型： 原生态数据类型和他们的包装类型（boolean除外）， `String` , `BigInteger` , `BigDecimal` , `java.util.Date` 和 `java.sql.Date` .

10.2. Configuration 配置

为了能够使用 JPA 的实体，引擎必须有一个对 `EntityManagerFactory` 的引用。 这可以通过配置引用或者提供一个持久化单元名称。作为变量的JPA 实体将会被自动检测并进行相应的处理

下面例子中的配置是使用 jpaPersistenceUnitName：

```
<bean id="processEngineConfiguration"
  class="org.activiti.engine.impl.cfg.StandaloneInMemProcessEngineConfiguration">

<!-- Database configurations -->
<property name="databaseSchemaUpdate" value="true" />
<property name="jdbcUrl" value="jdbc:h2:mem:JpaVariableTest;DB_CLOSE_DELAY=1000" />

<property name="jpaPersistenceUnitName" value="activiti-jpa-pu" />
<property name="jpaHandleTransaction" value="true" />
<property name="jpaCloseEntityManager" value="true" />

<!-- job executor configurations -->
<property name="jobExecutorActivate" value="false" />

<!-- mail server configurations -->
<property name="mailServerPort" value="5025" />
</bean>
```

接下来例子中的配置提供了一个我们自定义的 `EntityManagerFactory` (在这个例子中，使用了 OpenJPA 实体管理器)。注意该代码片段仅仅包含与例子相关的 beans， 去掉了其他 beans。OpenJPA 实体管理的完整并可以使用的例子可以在 activiti-spring-examples(`/activiti-spring/src/test/java/org/activiti/spring/test/jpa/JPASpringTest.java`)中找到。

```
<bean id="entityManagerFactory" class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
  <property name="persistenceUnitManager" ref="pum"/>
  <property name="jpaVendorAdapter">
    <bean class="org.springframework.orm.jpa.vendor.OpenJpaVendorAdapter">
      <property name="databasePlatform" value="org.apache.openjpa.jdbc.sql.H2Dictionary" />
    </bean>
  </property>
</bean>

<bean id="processEngineConfiguration" class="org.activiti.spring.SpringProcessEngineConfiguration">
  <property name="dataSource" ref="dataSource" />
  <property name="transactionManager" ref="transactionManager" />
  <property name="databaseSchemaUpdate" value="true" />
  <property name="jpaEntityManagerFactory" ref="entityManagerFactory" />
  <property name="jpaHandleTransaction" value="true" />
  <property name="jpaCloseEntityManager" value="true" />
  <property name="jobExecutorActivate" value="false" />
</bean>
```

同样的配置也可以在编程式创建一个引擎时完成，例如：

```
ProcessEngine processEngine = ProcessEngineConfiguration
.createProcessEngineConfigurationFromResourceDefault()
.setJpaPersistenceUnitName("activiti-pu")
.buildProcessEngine();
```

配置属性：

- jpaPersistenceUnitName: `使用持久化单元的名称（要确保该持久化单元在类路径下是可用的）。根据该规范，默认的路径是/META-INF/persistence.xml)。要么使用 jpaEntityManagerFactory` 或者 `jpaPersistenceUnitName` 。
- jpaEntityManagerFactory: `一个实现了javax.persistence.EntityManagerFactory 的 bean 的引用`。它将被用来加载实体并且刷

新更新。要么使用jpaEntityManagerFactory 或者jpaPersistenceUnitName。

- jpaHandleTransaction: 在被使用的EntityManager 实例上，该标记表示流程引擎是否需要开始和提交/回滚事物。当使用Java事物API（JTA）时，设置为false。
- jpaCloseEntityManager: `该标记表示流程引擎是否应该关闭从 EntityManagerFactory` 获取的 EntityManager的实例。当EntityManager 是由容器管理的时候需要设置为 false（例如 当使用并不是单一事物作用域的扩展持久化上下文的时候）。

10.3. Usage 用法

10.3.1. 简单例子

使用 JPA 变量的例子可以在 JPAVariableTest 中找到。我们将会一步一步的解释 `JPAVariableTest.testUpdateJPAEntityValues`。

首先，我们需要创建一个基于 `META-INF/persistence.xml` 的EntityManagerFactory 作为我们的持久化单元。它包含持久化单元中所有的类和一些供应商特定的配置。

我们将使用一个简单的实体作为测试，其中包含有一个 id 和 `String` 类型的 value 属性，这也将会被持久化。在允许测试之前，我们创建一个实体并且保存它。

```
@Entity(name = "JPA_ENTITY_FIELD")
public class FieldAccessJPAEntity {

  @Id
  @Column(name = "ID_")
  private Long id;

  private String value;

  public FieldAccessJPAEntity() {
    // Empty constructor needed for JPA
  }

  public Long getId() {
    return id;
  }

  public void setId(Long id) {
    this.id = id;
  }

  public String getValue() {
    return value;
  }

  public void setValue(String value) {
    this.value = value;
  }
}
```

我们开始一个新的流程实例，添加实体作为变量。与其它的变量一样，它们存储在引擎的持久存储区。当下次这个变量被请求，它会从基于类和Id的存储的 `EntityManager` 中加载。

```
Map<String, Object> variables = new HashMap<String, Object>();
variables.put("entityToUpdate", entityToUpdate);

ProcessInstance processInstance = runtimeService.startProcessInstanceByKey("UpdateJPAValuesProcess", variables);
```

在我们的流程定义的第一个节点包含一个 `serviceTask` 将调用方法 在 `entityToUpdate` 上 `setValue`，它解析为 JPA 变量，我们启动流程实例并将从相关联的当前引擎的上下文“EntityManager”进行加载。

```
<serviceTask id='theTask' name='updateJPAEntityTask'
  activiti:expression="${entityToUpdate.setValue('updatedValue')}" />
```

当 service-task 完成后，流程实例等在流程定义中定义的 userTask，这使我们能够检查流程实例。在这一点上，

EntityManager 已刷新并更改到实体已经被推到数据库。当我们得到变量 entityToUpdate 值时，它再次加载，我们得到实体，并将实体中的属性 `value` 设置到 `updatedValue` 。

```
// Servicetask in process 'UpdateJPAValuesProcess' should have set value on entityToUpdate.
Object updatedEntity = runtimeService.getVariable(processInstance.getId(), "entityToUpdate");
assertTrue(updatedEntity instanceof FieldAccessJPAEntity);
assertEquals("updatedValue", ((FieldAccessJPAEntity)updatedEntity).getValue());
```

10.3.2. 查询 JPA 处理变量

可以查询 `ProcessInstances` 和 `Execution` 包含 JPA 实体作为变量值。注意 只有 在 ProcessInstanceQuery 和 ExecutionQuery ， `variableValueEquals(name, entity)` 是支持 JPA 实体的 。 方法 `variableValueNotEquals` , `variableValueGreaterThan` , `variableValueGreaterThanOrEqual` , `variableValueLessThan` 和 `variableValueLessThanOrEqual` 不支持，并且当一个 JPA 实体传递作为值时，会抛出 `ActivitiException` 。

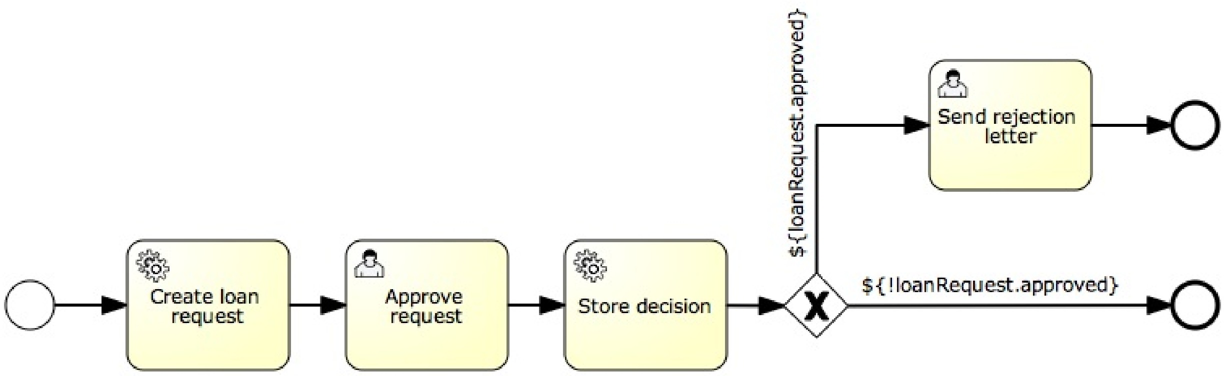
```
ProcessInstance result = runtimeService.createProcessInstanceQuery()
    .variableValueEquals("entityToQuery", entityToQuery).singleResult();
```

10.3.3. 高级例子：使用 Spring bean 和 JPA

JPASpringTest 例子可以在 activiti-spring-examples 中找到。它的使用场景如下：

- 存在一个使用 JPA 实体的 Spring bean，用于存储贷款请求。
- 使用 Activiti，我们可以利用那些由现有的 bean 获得的已有的实体，并将其作为变量在流程中使用。 流程是按以下步骤进行定义的：
  - 服务任务，利用已有的 LoanRequestBean 使用启动流程时接收的变量（比如，可以来源于开始的表单）来创建新的 LoanRequest。使用 activiti:resultVariable（它以一个变量来存储表达式结果）将创建出来的实体以变量进行存储。
  - 用户任务，让经理查看请求，批准/不批准，结果存储到一个 boolean 类型的变量 approvedByManager 上。
  - 服务任务，用来更新贷款请求实体以使该实体与流程同步。
  - 根据实体属性 approved 的值，利用排他分支来决定下一步选择哪条路径：当请求被批准时，流程将结束；否则，会另有一个任务（发送拒绝信），这样就可以由拒绝信手动地通知用户了。

请注意此流程不包含任何表单，所以其只用于单元测试




```
<?xml version="1.0" encoding="UTF-8"?>
<definitions id="taskAssigneeExample"
  xmlns="http://www.omg.org/spec/BPMN/20100524/MODEL"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:activiti="http://activiti.org/bpmn"
```

```
    targetNamespace="org.activiti.examples">

  <process id="LoanRequestProcess" name="Process creating and handling loan request">
    <startEvent id='theStart' />
    <sequenceFlow id='flow1' sourceRef='theStart' targetRef='createLoanRequest' />

    <serviceTask id='createLoanRequest' name='Create loan request'
      activiti:expression="${loanRequestBean.newLoanRequest(customerName, amount)}"
      activiti:resultVariable="loanRequest"/>
    <sequenceFlow id='flow2' sourceRef='createLoanRequest' targetRef='approveTask' />

    <userTask id="approveTask" name="Approve request" />
    <sequenceFlow id='flow3' sourceRef='approveTask' targetRef='approveOrDissaprove' />

    <serviceTask id='approveOrDissaprove' name='Store decision'
      activiti:expression="${loanRequest.setApproved(approvedByManager)}" />
    <sequenceFlow id='flow4' sourceRef='approveOrDissaprove' targetRef='exclusiveGw' />

    <exclusiveGateway id="exclusiveGw" name="Exclusive Gateway approval" />
    <sequenceFlow id="endFlow1" sourceRef="exclusiveGw" targetRef="theEnd">
      <conditionExpression xsi:type="tFormalExpression">${loanRequest.approved}</conditionExpression>
    </sequenceFlow>
    <sequenceFlow id="endFlow2" sourceRef="exclusiveGw" targetRef="sendRejectionLetter">
      <conditionExpression xsi:type="tFormalExpression">${!loanRequest.approved}</conditionExpression>
    </sequenceFlow>

    <userTask id="sendRejectionLetter" name="Send rejection letter" />
    <sequenceFlow id='flow5' sourceRef='sendRejectionLetter' targetRef='theOtherEnd' />

    <endEvent id='theEnd' />
    <endEvent id='theOtherEnd' />
  </process>

</definitions>
```

虽然上面的例子很简单，但确实展示出了结合 Spring 和参数化方法表达式来使用 JPA 的强大。此流程根本不需要定制 java 代码（当然了，除 Spring bean 外），大大加速了部署

CrystalBall inside 内部